Spring 5-20-2020

Rehearsal Scheduling Problem

Thuan Bao

Rehearsal Scheduling Problem

A Project

Presented to

The Faculty of Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Thuan Bao

March 2020

The Designated Project Committee Approves the Project Titled

Rehearsal Scheduling Problem

By

Thuan Bao

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

San Jose State University

May 2020

Dr. Robert Chun	Department of Computer Science
Dr. Chris Tseng	Department of Computer Science
Dr. Long Nguyen	Visa, Inc.

ABSTRACT

Scheduling is a common task that plays a crucial role in many industries such as manufacturing or servicing. In a competitive environment, effective scheduling is one of the key factors to reduce cost and increase productivity. Therefore, scheduling problems have been studied by many researchers over the past thirty years. Rehearsal scheduling problem (RSP) is similar to the popular resource-constrained project scheduling problem (RCPSP); however, it does not have activity precedence constraints and the resources' availabilities are not fixed during processing time. RSP can be used to schedule rehearsal in theatre industry or to schedule group scheduling when each member has different sets of available time. In this report, three different approaches are proposed to solve RSP including Constraint Programming, Integer Programming, and Schedule Generation Schemes.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to Dr. Robert Chun for his guidance and support throughout the entire project. It was my honor to have him as my advisor with his in-depth knowledge and expertise in the field.

I would like to thank my committee advisors, Dr. Chris Tseng and Dr. Long Nguyen for their valuable feedback and suggestions given towards the project. It was a great privilege to work with such dedicated and knowledgeable advisors.

Finally, I am grateful to my family and friends for always being a strong support system for me throughout my career.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

Rehearsal scheduling is a time consuming and tedious task in theatre industry. In theatre, a play usually has multiple scenes and for each scene, not all performers have to participate. Before practicing for the play, each performer will submit a set of available time to the scheduler. Based on these sets of available time and the duration needed to practice for each scene, a schedule is created with the effort to satisfy these constraints:

- A performer cannot practice more than one scene at the same time.
- Each scene's start time and end time must be in one of the available timeslots submitted by all performers who take part in that scene.

This scheduling problem is similar to an original resource-constrained project scheduling problem (RCPSP) with additional constraints of resource's availability but without precedence constraints. RCPSP is defined as a single project scheduling that satisfies the resource constraints with no preemption. All activity durations, resource requirements, and resource availabilities must be integral and known before scheduling [1]. The original RCPSP can be formulated as:

Optimize	Min t_n,
Subject to	$t_j - t_i \geq d_i$, $(i, j) \in H$,
	$\sum_{S(t)} r_{ik} \leq b_k$, $t = 1, \ldots, T$; $k = 1, \ldots, K$
Where	t_i = starting time of activity i, $i = l,\ldots,n$.
	H = set of precedence constraints.
	d_i = duration (processing time) of activity i.
	r_{ik} = amount of resource k that activity i requires.
	b_k = resource k's total availability.
	S(t) = the activities set being processed at time t

We observe that the rehearsal scheduling problem (RSP) is similar to the original RCPSP in term of activity and recourse modeling. RSP's practicing event can be considered as an activity and each performer of that event is similar to a resource required by that activity. However, activities in RSP can start in any order, i.e. no precedence constraints, and the resources required by each activity have the capacity of one. Furthermore, unlike the original RCPSP, resources in RSP are not available with a fixed amount during processing time. With this observation, we can formulate a RSP as:

Optimize	Min t_n,	
Subject to	Resource R must be available during $(t_i, t_i + d_i)$, $R \in R_i$	(1)
	$\sum_{S(t)} r_{ik} \leq 1$, $t = 1, \ldots, T$; $k = 1, \ldots, K$	(2)
Where	t_i = starting time of activity i, $i = 1,\ldots,n$.	
	R_i = list of resources required by activity i	
	d_i = duration (processing time) of activity i.	
	r_{ik} = amount of resource k required by activity i, either 0 or 1	
	S(t) = set of activities in process at time t	

Constraint (1) means that during the processing time of each activity, every resource required by this activity must be available. For example, if performers A and B participate in practicing event for scene 1, then this constraint can be described as resource A and B must be available during the processing time of activity 1. This is different from the original RCPSP that resources are always available with a fixed capacity at any processing time. Constraint (2) denotes the fact that all activities processed at the same time must not require the same resource, which has the capacity of either 0 or 1 at any processing time. An example of this constraint can be seen as if performer A takes part in both scene 1 and 2, then the practicing events of these two scenes must not be overlapping scheduled.

Blazewicz has shown that RCPSP is actually a generalization of the classic job shop scheduling problem (JSSP) [1]. In a job shop scheduling problem, there are *n* jobs that will be executed on *m* machines. The activities of each job must be executed in a specific order so that each activity requires a specific machine with a predefined processing time. The JSSP is a NP-hard problem which is known to be one of the most difficult in this class [2]. RSP does not have the constraint of precedence, but it has the additional constraint of resource's availability which makes the search space not smaller. Thus, we can consider RSP as an NP-hard problem as well.

During the past thirty years, many solutions have been proposed to solve RCPSP but not RSP. They can be classified into three categories including heuristic methods based on serial and parallel schedule generation schemes; exact method using branch-and-bound procedures; and metaheuristic methods based on Tabu search or genetic algorithm [1]. Based on the study of RCPSP and constrained scheduling problem in general, we propose three different approaches to solve RSP. In the first approach, we define the problem as a Constraint Satisfaction Problem (CSP) and then use a Constraint Programming solver to find the optimal solution. In the second approach, we model the problem as an Integer Programming problem, and then use a Mixed Integer Programming solver to find the optimal solution. The third approach is based on the serial Scheduling Generation Schemes technique with some heuristics to increase the runtime performance. We then conduct the performance benchmark for the three approaches with different criteria.

CONSTRAINT PROGRAMMING

1.1. Constraint Satisfaction Problem

1.1.1. Definition

Constraint Satisfaction Problem (CSP) is the fundamental concept in Constraint Programming. Assume we have a set of variables $Y = y_1,\ldots,y_k$ and a set of domains $D_1, \ldots, D_k$ so that each variable y_i has value in the domain D_i. Then a constraint C over Y is a subset of $D_1 \times \ldots \times D_k$. If $k = 1$ then the constraint is unary, if $k = 2$ then the constraint is called binary constraint. A CSP is defined as a sequence of variables $X = x_1, \ldots, x_n$ on domain $D_1, \ldots, D_n$, respectively so that a finite set C of constraints are specified with one or more subsequences of X [3]. Such CSP is notated as $\langle \mathcal{C}; \mathcal{DE} \rangle$ where $\mathcal{DE} = x_1 \in D_1, \ldots, x_n \in D_n$.

A CSP's solution is a sequence of valid variable values so that all the constraints are satisfied. Assuming we have a CSP denoted as $\langle \mathcal{C}; \mathcal{DE} \rangle$, the set $(d_1,\ldots, d_n) \in D_1 \times \ldots \times D_k$ satisfies the constraint $C \in \mathcal{C}$ on the variable $x_{i1},\ldots, x_{in}$ if $(d_{i1}, \ldots, d_{in}) \in C$ is a solution to $\langle \mathcal{C}; \mathcal{DE} \rangle$ if it satisfies all constraints in $\mathcal{C}$.

1.1.2. Example

1.1.2.1. Constraint Satisfaction Problem on integers

A well-known example of CSP on integers is the n queens problem. In this problem, the challenge is to arrange n queens on a chess board with side n x n so that there is no queen being attacked.

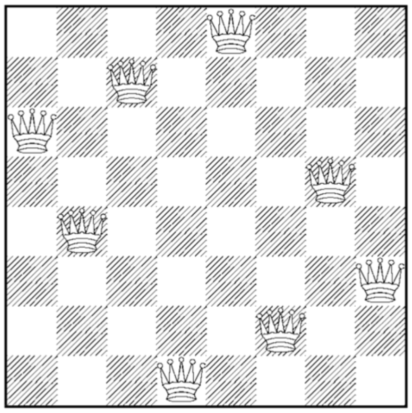

Figure 1. CSP example - 9 Queens Problem [3]

We can model this CSP by using n variables x_i with domain [1…n], i = 1,…,n so that x_i denote the position of the queen which is located at the ith column of the board. The constraints of this CSP can be conveyed through the following formula:

- $x_i \neq x_j$ (to satisfy the constraint that no two queens placed in the same row)
- $x_i - x_j \neq i - j$ (no two queens placed in South-West – North-East diagonal)
- $x_i - x_j \neq j - i$ (no two queens placed in North-West – South-East diagonal)
- $i \in [1, n-1]$ and $j \in [i + 1, n]$

1.1.2.2. Boolean Constraint Satisfaction Problem

We can represent the n queens problem in the different way using Boolean expression. Let's consider n^2 Boolean variables $x_{i,j}$ where $i,j \in [1,...,n]$ and each variable $x_{i,j}$ indicates if a queen is placed at row ith, column jth. Let one($s_1,…,s_k$) denote the Boolean expression that one and only one Boolean expression from the set of Boolean expressions [$s_1,…,s_k$] is true. Then the n queens problem can be represented as:

- one($x_{i,1}$, ..., $x_{i,n}$) for $i \in [1\ldots n]$ (satisfy the constraint of no queens placed at same row)
- one($x_{1,i}$, ..., $x_{n,i}$) for $i \in [1\ldots n]$ (satisfy the constraint of no queens placed at same column)
- not($x_{i,j}$ and $x_{k,l}$) for $i, j, k, l \in [1\ldots n]$, $i \neq k$ and $|i - k| = |j - l|$ (no 2 queens placed in the same diagonal)

1.1.3. Constrained Optimization Problem

Constrained optimization is a form of finding the optimal solution to a set of constraints subject to some objective functions [3]. Constrained optimization has been studied deeply, thus many researches have been conducted on this subject. One classic example of constrained optimization is the Knapsack problem. In this problem, there are *n* objects so that each object has a value and a volume, and a knapsack with fixed volume. We have to find a collection of objects so that the total value is maximized while all objects can be placed in the knapsack. We can model this problem by using *n* objects with volume $a_1,\ldots, a_n$ and values $b_1,\ldots,b_n$ while v will be the volume of the knapsack. Then we define n variables $x_1, \ldots, x_n$ with domain {0, 1} so that if object ith is put into the knapsack then $x_i = 1$. The requirement that objects fit in the knapsack can be formulated as $\sum_{i=0}^{n} a_i \times x_i \leq v$. The goal is to look for the solution that satisfies this constraint and the sum $\sum_{i=0}^{n} b_i \times x_i$ is maximized.

1.2. Constraint Programming

According to [3], constraint Programming is a programming process that is bounded to a set of constraints and a solution of these constraints by domain requirements or general methods [3]. We can solve a Constraint Programming problem by using a general approach with the help of constraint propagation.

1.2.1. Basic approach to solve a Constraint Programming problem

The first step to solve a Constraint Programming problem is to model the initial problem as a CSP. Modeling a CSP is more an art than science and usually requires rules of thumb or heuristics to have a good result [3]. The second step is to apply the generic procedure SOLVE defined in Figure 3 to the modeled CSP.

```
SOLVE:
VAR CONTINUE: BOOLEAN;
CONTINUE:= TRUE;
WHILE CONTINUE AND NOT HAPPY DO
  PREPROCESS;
  CONSTRAINT PROPAGATION;
  IF NOT HAPPY
  THEN
    IF ATOMIC
    THEN
      CONTINUE:= FALSE
    ELSE
      SPLIT;
      PROCEED BY CASES
    END
  END
END
```

Figure 2. Generic procedure SOLVE [3]

PREPROCESS: transform the original CSP to another form that is easier to process later.

HAPPY: this means all the constraints of the initial CSP has been satisfied. It is either a solution or all solutions have been found or an inconsistency has been detected.

ATOMIC: before we can split a CSP, we need to determine whether the current CSP can be split or not. This test is conducted by an ATOMIC procedure which will check if the CSP's domains are singleton sets or empty, or the search for another solution “under” this CSP is no longer needed.

SPLIT: if constraint propagation does not give us any solution which makes the test HAPPY failed and the current CSP $\mathcal{P}$ is not atomic, then we can split that CSP into two or more CSPs so that the union of them is equal to $\mathcal{P}$. Such split can be obtained by either splitting a domain or a constraint so that the split CSPs need to comply with a new set of constraints and domains.

PROCEED BY CASES: after splitting, there will be two or more new CSPs that will be considered in order. By using SPLIT procedure multiple times, a tree of CSPs will be formed. PROCEED BY CASES procedure will traverse this CSP tree in a specific order and keep track of newly added information at each node of the tree. The most well-known search techniques are backtracking, and branch and bound, which are applicable for searching optimal solutions [3].

CONSTRAINT PROPAGATION: this procedure will replace a given CSP by a simpler CSP but still guarantees the equivalence in term of logic satisfaction. “Simpler” refers to the fact that the new constraints or domains are reduced in term of size or search space. Constraint propagation is the most crucial and fundamental concept of Constraint Programming [4].

1.2.2. Constraint propagation algorithms

Constraint propagation algorithm deals with reduction of a domain or a constraint in a CSP. By using this algorithm, we can achieve a property called local consistency notation which is crucial for solving constraint programming problem. Local consistency is defined as for every variable x of constraint C, each value in x’s domain participates in an element of C [3]. This property is call hyper-arc consistency and in the case of binary constraints, it is called arc consistency. Details of constraint propagation can be found in [3], [5], [6], [2] and [7]

INTEGER PROGRAMING

2.1. Definition

According to [8], a pure integer program can be formulated as:

Max	cx
Subject to	$Ax \leq b$
	$x \geq 0$
Where	$c = (c_1,\ldots,c_n)$ is a row vector
	$A = m \times n$ matrix (a_{ij})
	$b = \begin{pmatrix} b_1 \\ . \\ . \\ b_m \end{pmatrix}$ is a column vector
	$x = \begin{pmatrix} x_1 \\ . \\ . \\ x_m \end{pmatrix}$ is a column vector that contains the variables to be optimized
	c, A, b contains the rational values while x contains integer values

A mixed integer program is more relaxing than a pure integer program and can be formulated as:

Max	$cx + hy$
Subject to	$Ax + Gg \leq b$
	$x \geq 0,\ y \geq 0$
Where	$c = (c_1,\ldots,c_n)$, $h = (h_1,\ldots,h_n)$ are a row vectors
	$A = m \times n$ matrix (a_{ij}), $G = m \times p$ matrix (g_{ij})

b = $\begin{pmatrix} b_1 \\ . \\ . \\ b_m \end{pmatrix}$ is a column vector

x = $\begin{pmatrix} x_1 \\ . \\ . \\ x_m \end{pmatrix}$, y = $\begin{pmatrix} y_1 \\ . \\ . \\ y_m \end{pmatrix}$ are column vectors that contains the variables to be optimized

c, A, b, y contains the rational values while x contains integer values

Example of mixed integer and pure integer programs can be found in Figure 4. In this example, the solution set of the mixed integer program is (x,y) where x is integral and y is rational. In contrast, the pure integer program has the solution set of concrete integers.

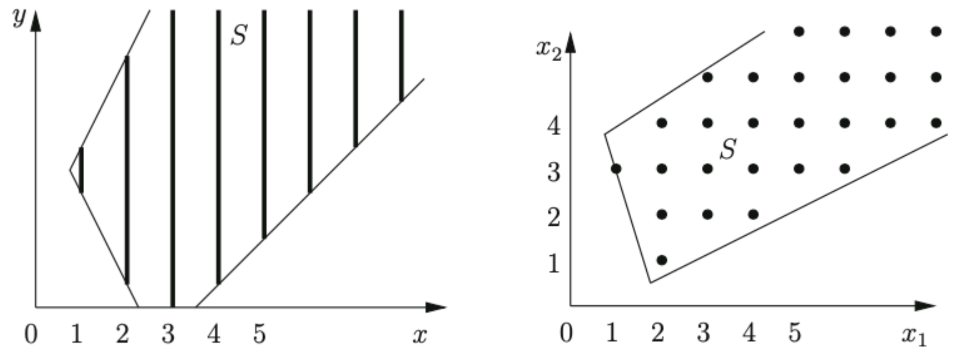

Figure 3. Example of mixed integer and pure integer [8]

2.2. Integer Programming Methods

Branch-and-bound and cutting plane methods are the two algorithmic principles for solving integer programs [8]. The mixed integer program mentioned above can be rewritten to be easier for reference as:

$$\text{MILP:} \quad \max \{cx + hy : (x, y) \in S\} \tag{2.1}$$
$$\text{where } S := \{(x, y\} \in Z_+^n \times R_+^p : Ax + Gy \leq b\}$$

We denote (x^*, y^*), z^* as the optimal solution and the optimal value of the above mixed integer program, respectively. (x^0, y^0) and z_0 are the optimal solution and optimal values of linear programming relaxation

$$\max \{cx + hy : (x, y) \in P_0\}$$

where P_0 is the linear relaxation of S. Because $S \subseteq P_0$, it is easy to conclude that $z^* \leq z_0$. When x^0 is an integral vector, we will have $(x^0, y^0) \in S$ and $z^* = z_0$ will be the optimal value of S.

2.2.1. Branch-and-Bound method

Assuming we have an index j where $1 \leq j \leq n$ so that x_j^0 is fractional and let $f := x_j^0$.

Define two sets

$$S_1 := S \cap \{(x, y): x_j \leq \lfloor f \rfloor\}$$

$$S_2 := S \cap \{(x, y): x_j \geq \lceil f \rceil\}$$

where $\lfloor f \rfloor$ is the largest integer $k \leq f$ and $\lceil f \rceil$ is the smallest integer $l \geq f$. If x_i is an integer for each $(x,y) \in S$ then it derives that (S_1, S_2) is a partition of S. Now we have two integer programs as:

$$\text{MILP1:} \quad \max\{cx + hy : (x, y) \in S_1\}$$

$$\text{MILP2:} \quad \max\{cx + hy : (x, y) \in S_2\}$$

The optimal solution of MILP described in (2.1) is the best between the optimal solutions of MILP1 and MILP2. Thus, finding the optimal solution of the original problem is now considered as finding the solutions for the two new subproblems.

Let call P_1, P_2 the natural linear relaxation of S_1, S_2 so that $P_1 := P_0 \cap \{(x, y) : x_j \leq \lfloor f \rfloor\}$ and $P_2 := P_0 \cap \{(x, y) : x_j \leq \lceil f \rceil\}$ and consider the two linear relaxation programs LP1: $\max\{cx + hy : (x, y) \in P_1\}$ and LP2: $\max\{cx + hy : (x, y) \in P_2\}$. Here are the steps for finding the optimal solution:

(i) If one of the LP_i linear program is infeasible the S_i is infeasible as well because $S_i \subseteq P_i$. Therefore, $MILP_i$ is infeasible and the algorithm stops here.

(ii) Let (x^i, y^i) be the optimal solution of LP_i and z_i be its value, $i = 1, 2$

a. If x^i is an integral vector, it derives that (x^i, y^i) is an optimal solution of $MILP_i$ and then a solution of MILP. Because of $S_i \subseteq S$, we can conclude $z_i \leq z^*$ and z_i becomes a lower bound of the optimal value of MILP.

b. If x^i is not an integral vector and z_i is not greater than the best known lower bound of the MILP then S_i cannot have any better solution than the current lower bound.

c. If x^i is not an integral vector and z_i is greater than the best known lower bound of the MILP then S_i might still have a better solution than the current lower bound. Let $x^i_{j'}$ be a fractional component of vector x^i, $f' := x^i_{j'}$ then create the sets $S_{i_1} := S_i \cap \{(x,y) : x_{j'} \leq \lfloor f' \rfloor\}$, $S_{i_2} := S_i \cap \{(x,y): x_{j'} \geq \lceil f' \rceil\}$ and repeat the above process until the program stops.

The procedure above finds the optimal solution by partitioning the set S into subsets and limits the enumeration by bounding the objective value of the subproblems [8]. It generates a list of linear programming problems achieved by relaxing the integral requirements on the variables x_j and establishes bounds on these variables.

2.2.2. Cutting Plane method

Assuming we have an integer program as:

MILP: $\max \{cx + hy : (x, y) \in S\}$

where $S := \{(x, y\} \in Z_+^n \times R_+^p : Ax + Gy \leq b\}$

Denote z_0 and (x_0, y_0) as the optimal value and solution of the above MILP's linear relaxation program formulated as $\max \{cx + hy : (x, y) \in P_0\}$. The main idea of cutting plane method is to find an inequality $\alpha x + \gamma y \leq \beta$ which is met by all points in S but not (x_0, y_0). A valid inequality $\alpha x + \gamma y \leq \beta$ for S that is not satisfied by (x_0, y_0) is a cutting plane separating (x_0, y_0) from S [8].

Let $\alpha x + \gamma y \leq \beta$ be a cutting plane and define $P_1 := P_0 \cap \{(x, y) : \alpha x + \gamma y \leq \beta$. Because of $S \subseteq P_1 \subset P_0$, the linear programming relaxation of MILP based on P_1 is stronger than the natural linear programming relaxation. The cutting plane method is based on this observation with the recursive approach [8]:

Initialize $i = 0$, repeat:

Solve linear program of $\max \{cx + hy : (x, y) \in P_i\}$

- If S consists the optimal solution (x^i, y^i), then stop.
- Else solve the problem:
    - Find new cutting plane $\alpha x + \gamma y \leq \beta$ so that (x^i, y^i) is separated from S.
    - Assign $P_{i+1} := P_i \cap \{(x, y) : \alpha x + \gamma y \leq \beta\}$, then repeat the above steps

SCHEDULING PROBLEM

3.1. Scheduling theory

3.1.1. Definition

Scheduling problem is identified as a problem of allocating scarce resources to activities over time [2]. Activities are denoted as $\{A_1, \ldots, A_n\}$ while resources are denoted as $\{R_1, \ldots, R_n\}$. Each activity has a processing time and demands a certain capacity from one or many resources. Each resource has a fixed capacity that cannot be overloaded at any time. A resource with capacity of one is named as a machine. There will be a set of temporal constraints between activities and a cost function. The goal of scheduling problem is to determine when each activity is executed so that the overall cost is minimized, and all constraints are satisfied.

Based on resource type, scheduling problem can be classified into disjunctive scheduling and cumulative scheduling. For disjunctive scheduling problem, each resource is a machine so it can execute one activity at a time at most. Contrarily, a resource can execute multiple activities at the same time in a cumulative scheduling problem as long as the resource capacity is not exceeded.

Based on activity type, scheduling problem can be grouped to non-preemptive scheduling, preemptive scheduling and elastic scheduling. Activities in non-preemptive scheduling problem are not allowed to be interrupted. Preemptive scheduling activities, in contrast, can be interrupted so that other activities can execute using the same resource. In elastic scheduling, resource's amount allocated to an activity can have any value between 0 and the resource's capacity. However, there is another constraint as the sum over time of the allocated capacity must equal to a predefined value, which is denoted as energy.

We can also classify scheduling problems into decision problems and optimization problems. The goal of decision problem is to find an existing schedule that can satisfy all constraints. In optimization problems, the goal is to find a satisfied schedule that is optimal using objective function.

3.1.2. Examples

Pinedo et al. [9] has given examples of some classic scheduling problems as:

- Flow Shop: there are *m* machines and each job are processed on one of these machines. All jobs must be processed in the same route, for example, they are processed on machine 1 first, then on machine 2, etc. The objective is to find the order to execute all jobs so that the sum of completion time is minimized.
- Job Shop: a job shop problem has *m* machines and each job has different predetermined route to follow. The objective of this problem is to minimize the makespan, which is the finished time of the latest job. This problem is considered the fundamental of scheduling literature, thus it has been studied widely by many researchers.
- RCPSP: a project scheduling problem that is generalized based on job shop problem [1]. RCPSP is similar to RSP and has been mentioned in the report's introduction.

3.1.3. Constraint-Based Scheduling Model

3.1.3.1. Activity

In non-preemptive scheduling problem, we use three variables to represent an activity A_i as $proc(A_i)$, $start(A_i)$, and $end(A_i)$ that denote the processing time, start time, and end time. If we denote r_i and d_i as the release time and the deadline of activity A_i then $[r_i, d_i]$ is the time period in which A_i must be executed. The domains of $start(A_i)$ and $end(A_i)$ will be $[r_i, lst_i]$ and $[eet_i, d_i]$ so that *lst* stands for latest start time and *eet* stands for earliest end time. Activity A_i's processing time is calculated as $proc(A_i) = end(A_i) - start(A_i)$. Figure 4 demonstrates an example of an activity model. Light grey color denotes the time window $[r_i, d_i]$ and dark grey color denotes the activity's processing time.

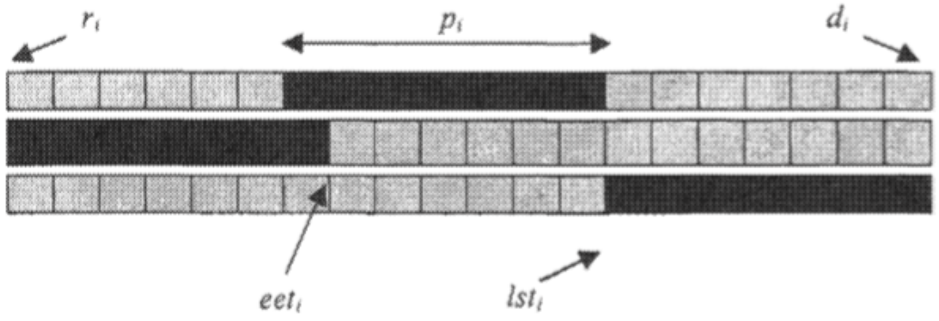

Figure 4. Activity modeling [3]

3.1.3.2. Resource Constraints

Resource constraints express the requirement that an activity requires a certain amount of resources throughout its execution. For activity A_i and resource R, let *cap*(R) denote the capacity of resource R and $cap(A_i, R)$ denote the amount of resource R required by activity A_i. If we denote $E(A_i, R)$ as the energy needed by activity A_i upon resource R, then the equality $E(A_i, R) = cap(A_i, R) * proc(A_i)$ is true for non-elastic problem. If we denote the unit amount of resource R required by activity A_i during the processing time t as $E(A_i, t, R)$, then the following constraint must be satisfied for every activity:

$$E(A_i, R) = \sum_t E(A_i, t, R)$$

And for each time t, this constraint represents the fact that the capacity of resource R cannot be exceeded:

$$\sum_{i=1}^{n} \mathrm{E}(\mathrm{A}_i, t, R) \leq cap(R)$$

In the non-preemptive scheduling problem, the above expression can be rewritten as:

$$\sum_{A_i:\, start(A_i) \leq t < end(A_i)}^{n} \mathrm{cap}(\mathrm{A}_i, R) \leq cap(R)$$

3.1.3.3. Objective Function

Objective function is used to find the optimal solution of a scheduling problem. Usually, the objective function takes in a set of end variables of the activity [2]:

criterion = F(end(A_1},…, end(A_n))

To find the optimal solution of a problem, we can solve the problem's successive decision variants. We can iterate through the possible values from the lower bound to upper bound of the domain, or vice versa. More details of objective functions can be found in [2], [5].

3.2. Constraint Propagation of Scheduling Problem

According to [10], a constrained scheduling problem usually includes:

- Temporal constraints including activities' possible values of start and end time.
- Resource constraints which define the activity's requirement of resource sets and the corresponding capacity.
- Capacity constraints which limit each resource's available capacity over time.
- Problem-specific constraints that address particular features of the activities and resources. These features will vary from problem to problem.

Let consider a simple disjunctive non-preemptive scheduling problem, which can be defined as a set of n activities {A_i, …, A_n} requiring the same resource with the capacity of one. Propagation on cumulative and preemptive is more complex and can be found in [2], [5].

3.2.1. Time-Table Constraint Propagation

Resource constraints in non-preemptive problem can be propagated by using Time-Table method to maintain resource utilization and availability information. Resource constraints can be propagated in two ways, from resources to activities or from activities to resources. Resources to activities propagation will update the activities' time bounding, i.e., earliest start times and latest end times, based on the resources' availability. Activities to resources propagation can update the Time-Tables based on the activities' time bounds [2]. The propagation is the process of maintaining an arc-consistency for any time t with the expression:

$$\sum_{i=1}^{n} E(\mathrm{A}_i, t) \leq 1$$

Let denote $X(\mathrm{A}_i, t)$ as a formula that has the value of 1 when activity A_i is executed at time t, 0 otherwise. Then $X(\mathrm{A}_i, t) = 1$ only when $start(\mathrm{A}_i) \leq t < end(\mathrm{A}_i)$. For this particular problem, we will have $E(\mathrm{A}_i, t) = X(\mathrm{A}_i, t)$ because the capacity for the resource is 1. We can propagate further with r_i, eet_i, lst_i and d_i as:

$$start(\mathrm{A}_i) \geq \min \{t: ub(X(\mathrm{A}_i, t) = 1\}$$

$$\mathrm{end}(\mathrm{A}_i) \leq \max \{t: ub(X(\mathrm{A}_i, t)) = 1\} + 1$$

$$[X(\mathrm{A}_i, t) = 0] \wedge [t < eet_i] \Rightarrow [start(\mathrm{A}_i) > t]$$

$$[X(\mathrm{A}_i, t) = 0] \wedge [lst_i < t] \Rightarrow [end(\mathrm{A}_i) < t]$$

Before Propagation	r_i	d_i	p_i
A_1	0	3	2
A_2	0	4	2
Propagation 1	r_i	d_i	p_i
A_1	0	3	2
A_2	2	4	2
Propagation 2	r_i	d_i	p_i
A_1	0	2	2
A_2	2	4	2

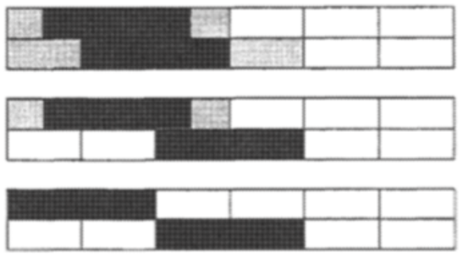

Figure 5. Time-Table constraint propagation [3]

Figure 5 demonstrates the propagation of two activities requiring the same resource with the capacity of one. A_1's latest start time (($lst_1 = d_1 - p_1 = 1$) is smaller than its earliest end time ($eet_1 = r_1 + p_1 = 2$). This informs that A_1 can only be executed between 1 and 2. Throughout this period, $X(A_1, t)$ is equal to 1 so that the related resource is not available for A_2 anymore. Because A_2 is not allowed to be interrupted or to finish before 1, its earliest start and end times are changed to 2 and 4 by propagation 1. This will enforce $X(A_2, t)$ to be set to 1 during the period [2, 4], thus A_1's latest end time will be updated to 2 by propagation 2.

3.2.2. Edge-Finding Propagation

The term "Edge-Finding" represents both a "branching" and a "bounding" technique [2]. Branching technique is to order activities which require the same resource. In branching technique, a set of activities Ω is chosen at each node, and for each activity $A_i \in \Omega$, a new branch is generated with the constraint that A_i is executed first (or last) in Ω. Bounding technique is used to derive that some activities of set Ω must, can, or cannot be executed first (or last) in Ω. These observations will indicate new time bounds and new ordering relations so that activities' earliest start times and latest end times can be strengthened.

Let r_Ω and $eetmin_\Omega$ represent the smallest of the earliest start time and smallest of the earliest end time of all activities in Ω respectively. Similarly, let $lstmax_\Omega$ and d_Ω represent the largest of the latest start times and the largest of the latest end times of all activities in Ω, respectively. Let p_Ω denote the sum of all activities' minimal processing times in Ω, $A_i << A_j$ (or $A_i >> A_j$) denote A_i executes before (or after) A_j, and $A_i << \Omega$ (or $A_i >> \Omega$) denote that A_i executes before (or after) all activities in Ω. Then the following rules can be applied to the Edge-Finding bounding technique [2]:

$$\forall\Omega, \forall A_i \notin \Omega, [d_{\Omega \cup \{A_i\}} - r_\Omega < p_\Omega + p_i] => [A_i << \Omega]$$

$$\forall\Omega, \forall A_i \notin \Omega, [d_\Omega - r_{\Omega \cup \{A_i\}} < p_\Omega + p_i] => [A_i >> \Omega]$$

$$\forall\Omega, \forall A_i \notin \Omega, [A_i << \Omega] => [end(A_i) \leq \min_{\varnothing \neq \Omega' \subseteq \Omega} (d_{\Omega'} - p_{\Omega'})]$$

$$\forall\Omega, \forall A_i \notin \Omega, [A_i >> \Omega] => [start(A_i) \geq \max_{\varnothing \neq \Omega' \subseteq \Omega} (r_{\Omega'} + p_{\Omega'})]$$

3.3. Schedule Generation Schemes for RCPSP

Scheduling Generation Schemes (SGS) is the fundamental of most heuristic solutions used to solve a RCPSP [1]. There are two types of SGS, distinguished by the usage of activity-incrementation and time-incrementation. Serial SGS employs activity-incrementation while parallel SGS employs time-incrementation. Our approach to solve the scheduling problem is based on serial SGS so we will describe it here. The concept of parallel SGS can be found in section 7.2 of [1].

Let assume that the RCPSP has n activities that need to be scheduled. Serial SGS will consist of *n* stages *g* so that in each stage, an activity is chosen and scheduled at the earliest precedence and resource-feasible completion time. At each stage *g*, two set of activities S_g and D_g are presented so that S_g includes all scheduled activities while D_g includes all activities available for scheduling. One important note is that the union of D_g and S_g is not equal to the activities set because there might be some activities that are not available to be scheduled at stage g due to precedence constraints. Let $\tilde{R}_k = R_k - \sum_{j \in A(t)} r_{j,k}$ denote the remaining capacity of resource *k* during processing *t* and let $F_g = \{F_j \mid j \in S_g\}$ denote finish times set. According to [1], the algorithm for serial SGS is given as:

```
Initialization: $F_0 = 0, S_0 = \{0\}$,
For g = 1 to n do:
    Calculate $D_g, F_g, \tilde{R}_k(t)$ $(k \in K, t \in F_g)$
    Select one $j \in D_g$ so that:
        $EF_j = max_{h \in P_j}\{F_h\} + p_j$
        $F_j = min\ \{t \in [EF_j - p_j, LF_j - p_j]\} \cap F_g$
        $r_{j,k} \leq \tilde{R}_k(t),\ k \in K,\ r \in [t, t + p_j[\cap F_g\} + p_j$
    $S_g = S_{g-1} \cup \{j\}$
$F_{n+1} = max_{h \in P_{n+1}}\{F_h\}$
```

The fake activity j = 0 is initially assigned with a completion time of 0 and is added to the partial schedule. At each stage the set of finish times F_g, the decision set D_g and the remaining capacity of resource $\tilde{R}_k(t)$ are calculated. Then an activity *j* is chosen from the decision set. Activity *j*'s finish time is calculated by the earliest precedence feasible finish time EF_j and the earliest resource-feasible time F_j in $[EF_j, LF_j]$ [1].

According to [1], serial SGS always produces a feasible scheduler if the resources are unconstrained and the time complexity of above algorithm is O $(n^2 * K)$

REHEARSAL SCHEDULING PROBLEM

4.1. Problem Definition

As mentioned in the introduction section, RSP is similar to RCPSP but without activity precedence constraints and resources' availabilities are not fixed at processing time. An example of this scheduling problem is described in Table 1. Scene E1, i.e. event E1, has the duration of 90 minutes while E2 and E3 have the duration of 60 minutes. Performers P1 and P2 have to attend the event E1, while the event E2 requires P1, P2, P4 and the event E3 requires P3, P4. All performers have submitted their availabilities during the period 01/01 – 01/05.

	E1 (90)	E2 (60)	E3 (60)	01/01	01/02	01/03	01/04	01/05
P1	x	x		8 am - 11 am 2 pm – 5 pm	8 am – 11 am 3 pm – 5 pm	None	None	8 am – 10 am
P2	x	x		8 am – 5 pm	12 pm – 1 pm	None	3 pm – 5 pm	8 am – 10 am
P3			x	2 pm – 3 pm	None	None	12 pm – 3 pm	9 am – 11 am
P4		x	x	None	8 am – 5 pm	12 pm – 1 pm	12 pm – 1 pm	9 am – 12 pm

Table 1. Rehearsal example

Usually, there are more than one solution that can satisfy all the constraints in a RSP. The goal is to find the solution having the minimum makespan, which is the time when the last event finishes. An optimal solution for the above example is demonstrated in Figure 8. The timeslot with blue color is where the event E1 occupies. The event E2 occupies timeslot with orange color, while the event E3 occupies magenta timeslot. Green-color timeslots are where the participant is still available for any event. Schedule in Figure 6 is optimal because there is no other schedule that can satisfy all the constraints and has a finish time of the last event earlier than 01/05 10:00.

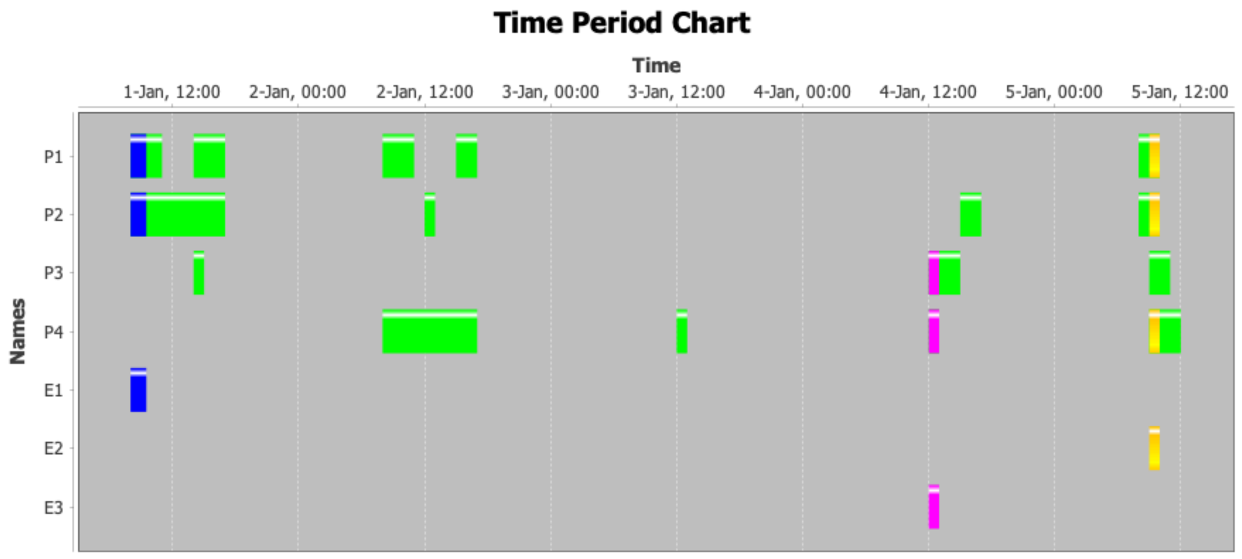

Figure 6. Optimal solution of RSP example

The RSP can be used to solve rehearsal scheduling problems in theatre industry or group scheduling with the similar time requirements. To the best of our knowledge, there is no direct research on this particular problem. In this report, we propose three approaches to solve RSP using Constraint Programming, Integer Programming and serial Schedule Generation Schemes technique. Then we do experiment on these approaches with different test criteria.

4.2. Constraint Programming approach

4.2.1. Problem model

The crucial step of this approach is to model the scheduling problem as a CSP so that a constraint programming solver can interpret it correctly. We observe that the original problem can be transformed to a CSP without losing any logic by applying the Time-Table technique described in section 3.2.1. More specifically, we get the list of all participants of each event and then based on the availability of each participant, we can calculate the possible timeslots that each event can occupy. Figure 7 displays the result of the original problem after this propagation step.

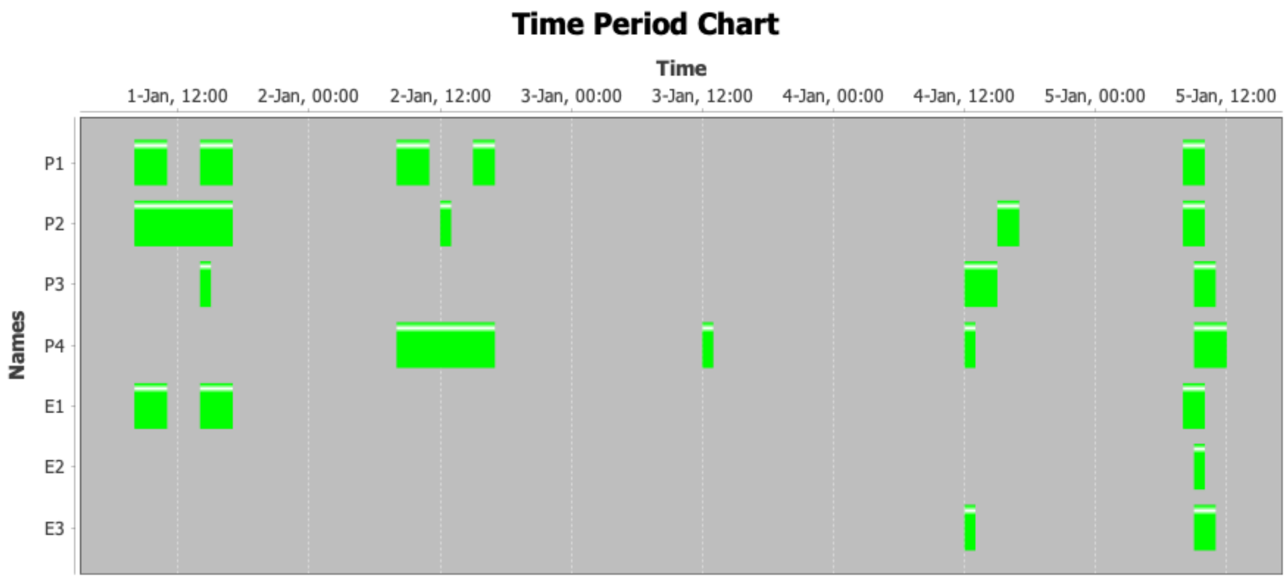

Figure 7. Time-Table propagated problem

The event E1 requires P1 and P2, therefore after being Time-Table propagated, its possible timeslots will be [01-01 08:00, 01-01 11:00], [01-01 14:00, 01-01 17:00] and [01-05 08:00, 01-05 10:00]. We can consider the possible timeslots of E1 as the intersection between the available timeslots of P1 and P2. We can calculate the possible timeslots of E2 and E3 using the same approach, and the final result is displayed in Table 2.

	01/01	01/02	01/03	01/04	01/05
E1	08:00 - 11:00 14:00 - 17:00				08:00 - 10:00
E2					09:00 - 10:00
E3				12:00 - 13:00	09:00 - 11:00

Table 2. Events' possible slots after Time-Table propagation

After this step, each event will have a list of possible timeslots. This will guarantee that if an event is scheduled at one of its possible timeslots, all of its participants are able to attend. But it is not guaranteed that there is no overlap between events which require at least one same participant. In our example, the event E1 must not overlap with the event E2 because P1, P2 participate in both events. Similarly, E2 and E3 must not overlap because they both require P4 while E1 and E3 do not have this constraint.

With these observations, we can model our problem (after doing Time-Table propagation) with a CSP as:

- Each event consists of a list of possible timeslots
- Events that require at least one same participant must not be overlapped.

4.2.2. Problem solver

At this step, we will choose a constraint programming solver to solve the above CSP model. There are many constraint programming solvers developed over the past years such as Choco 4, JaCop, Yuck or Google OR-Tools. In this report, we use Google OR-Tools [11] as our primary solver for both Constraint Programming and Integer Programming approaches. Below is the pseudo code of the problem solved by OR-Tools:

```
Start:                  Do Time-Table propagation on the original problem
For each event $E_i$:
                        Create a time interval variable ($start_i$, $start_i + d_i$)
                        For each available timeslot ($tstart_{ik}$, $tend_{ik}$) of $E_i$, create a Boolean
                        variable $ef_{ik}$ and add the constraints:
                            • $start_i \geq tstart_{ik}$ only enforced if $ef_{ik}$ is true          (4.1)
                            • $start_i + d_i \leq tend_{ik}$ only enforced if $ef_{ik}$ is true.     (4.2)
```

```
                              Add a XOR constraint of all $ef_{ik}$ variables                 (4.3)
For each pair of events
($E_i$, $E_j$) that require at
least one same
participant:                  Add not overlapped constraint between ($start_i$, $start_i + d_i$)
                              and ($start_j$, $start_j + d_j$)                               (4.4)
Add objective function:       Minimize max {$start_i + d_i$}, $i = 1, \ldots, n$              (4.5)
```

The challenge of this approach is how we can programmatically model the constraint of start time and end time of each event so that these values must be in one of the available timeslots. By introducing a new Boolean variable ef_{ik}, we tell the solver to only enforce constraints (4.1) and (4.2) if this variable is set to true. This will add m x n variables to the problem, where m is the average available timeslots per event and n is the number of the events. Constraint (4.3) will make sure that only one variable from the set of ef_{ij} is set to true, then it will enforce the start time and end time of event i to fall into one and only one possible timeslot. Constraint (4.4) tells the solver that the interval of events sharing at least one same participant cannot be overlapped. Then we add an objective function at (4.5) to minimize the makespan of the schedule. The complexity of this algorithm is calculated by the variables used, which is derived to the above m x n expression because the number of other used variables is relatively small compared to this value.

Once the problem is modeled so that the solver can understand all constraints, the heavy work to find the optimal solution will be loaded to the solver. Overall, constraint programming approach gives a good performance on RSP because the requirements of RSP fit naturally with a Constraint Programming model.

4.3. Integer Programming approach

4.3.1. Problem model

It is difficult to model the RSP as an integer programming problem without extra work. Lorraine et al. (2006) had successfully used integer programming to solve the nurse scheduling problem, which is another classic scheduling problem [12]. By looking into their work, we can have an overview of how to solve a scheduling problem using Integer Programming.

A simple nurse scheduling problem can be described as a scheduling problem in hospital subject to these conditions:

- There are three shifts in each day.
- Each nurse is responsible for each shift and no nurse works on more than one shift per day.
- During a specific period, each nurse is assigned at least two shifts.

In their paper, Lorraine et al. studies the anesthesiology nurse scheduling problem, which is more complicated than the above nurse scheduling problem. In addition, the RSP is not similar to scheduling problem. However, we can borrow the idea of using binary variables to enforce an event to start at a particular time.

With the same observation in section 4.2.1, we do Time-Table propagation on the RSP first. Then the original problem is transformed to a new problem that is illustrated in Figure 7. However, the new problem is not ready for integer programming yet. At this stage, the possible timeslot of each event, which is a pair of start time and end time, is still a continuous range. We need to do an extra step to map these continuous ranges into sets of concrete integral values so that an integer programming solver can understand. Basically, a timeslot is a pair of start time and end time, so to convert these pairs to integer we can use UNIX timestamp. UNIX timestamp is calculated by the number of seconds that have been spent since January 1, 1970. We can do even better by converting the time value based on the minimum start time of all available timeslots. This can help reduce the computational space, which can boost the runtime performance. At this step, possible timeslot is modeled as a pair of (start, end) so that start, end is integral. If we minus the end time by the event's duration and create a new pair as (start, end – duration) then the new pair deals with the start variable only.

Unfortunately, these new pairs are still not ready for integer programming because not every integer between each pair can be a candidate of the event's start time. Let's assume that each event's duration is a multiple of 30, and the available time submitted by each participant is rounded to hour or half hour. With this assumption, we divide the value of each pair by 30 then each integer number between each pair can be a valid start time of the event. Finally, the original RSP is transformed so that we can apply integer programming as:

- Each event consists a list of integer pairs. The start time of the event will be an integer between a certain pair.
- Events require at least one same participant must not be overlapped

According to Table 1, the earliest start time of all events is 01/01 08:00 so we can use this value as the lower bound. We then convert all the available timeslots to UNIX timestamp, and minus those values by this lower bound. For example, the first available timeslot of event E1 is 01/01 08:00 – 01/01 11:00, then after converting and subtracting, this timeslot can be re-written as 0 - 10,800. E1 has the duration of 90 minutes so if E1 is scheduled during this timeslot then the start time of E1 must be lesser than 10,800 – 90 * 60 = 5400, or E1's start time is ranged between 0 and 5400. We assume that each event's duration is a multiple of 30, so if we divide this pair by 1800, i.e. 30 minutes calculated in seconds, then the start time of E1 can be any integer values between 0 and 3. We keep applying these steps to all available timeslots of all events in Table 1, then the original integer programming problem can be stated as:

Minimize the makespan

Subject to:

$\text{start(E1)} \in Z^{+}$, start(E1) ∈ [0,3] v start(E1) ∈ [12,15] v start(E1) ∈ [192,193]

$\text{start(E2)} \in Z^{+}$, start(E2) ∈ [194,194]

$\text{start(E3)} \in Z^{+}$, start(E3) ∈ [152,152] v start(E3) ∈ [194,196]

E1, E2 and E2, E3 are not overlapped

Where:

start(Ei) is the start time of event Ei, i = 1, …,3

start(Ei) ∈ [x, y] means start time of event Ei is an integer value between x and y, inclusively

4.3.2. Problem solver

After the preparation steps in section 4.3.1, the problem can be modeled so that an integer programming solver can understand. The pseudo code is described as below:

```
Start:
        Do Time-Table propagation on the original problem
        Convert each event's possible timeslots to pairs of integer numbers
        LI: a large enough integer number in the search space                (4.6)
        Define objective variable objVar so that it can be minimized
        Define integer variable maxVar and then set objVar = maxVar
For each event $E_i$:
        For each possible slot ($start_{ij}$, $end_{ij}$) of $E_i$:
                Create integer variable $startVar_i$
                Create Boolean variable $logicalVar_{ij}$
                Add two inequalities:
```

```
            • $startVar_i - logicalVar_{ij} * LI \geq start_{ij} - LI$        (4.7)
            • $startVar_i + logicalVar_{ij} * LI \leq end_{ij} + LI$          (4.8)
        Add equality: $\sum_j logicalVar_{ij} = 1$                            (4.9)
        Create an integer variable $maxVar_i$ in range (0, LI)
        Add equality: $maxVar - startVar_i - d_i = maxVar_I$                  (4.10)
    For each pair of events ($E_i$, $E_j$) that require at least one same participant:
        Create Boolean variable $boolVar_{ij}$
        Add two inequalities:
            • $startVar_i - startVar_j + boolVar_{ij} * LI \geq d_j$           (4.11)
            • $startVar_i - startVar_j + boolVar_{ij} * LI \leq LI - d_i$      (4.12)
```

The LI constant in (4.6) is actually a "large enough" integer number so that the solver can interpret it as an upper bound for any computational inequality. The three constraints (4.7) to (4.9) are used to enforce that the start time of event E_i must be a valid integer value among its pairs. LogicalVar_{ij} is a Boolean variable so its value is either 0 or 1. If logicalVar_{ij} is equal to 0 then inequalities (4.7) and (4.8) are naturally satisfied because LI is a large integer number. Otherwise, when logicalVar_{ij} is equal to 1 then from these two constraints we can derive that $\text{start}_{ij} \leq \text{startVar}_i \leq \text{end}_{ij}$. This inequality expression will imply that the start time of event E_i is a valid value. Equality (4.9) will make sure only one logicalVar_{ij} variable is set to 1, which enforces the start time of event E_i must be a valid value of one and only one pair. Equality (4.10) is used to express the fact that *maxVar* is the maximum value between all events' end time.

Inequalities (4.11) and (4.12) are used to make sure that the processing period of two events that share at least one participant are not overlapping. Again, boolVar_{ij} is a Boolean variable so its value is either 0 or 1. If boolVar_{ij} is equal to 0 then from (4.11) we have $\text{startVar}_i - \text{startVar}_j \geq d_j$. This will ensure that event E_i is scheduled after event E_j is finished. (4.12) is naturally satisfied in this case because LI is a large enough integer number. If boolVar_{ij} is set to 1 then (4.12) is derived to $startVar_i - startVar_j \leq d_i$, which refers that event E_j is happening after E_i is finished. Inequality (4.11) is satisfied naturally if $boolVar_{ij}$ is equal to 1 thanks to the fact that LI is a large integer number.

We can see that solving RSP using Integer Programming approach is not as simple as using Constraint Programming approach. With a large test input, Integer Programming approach will require a very large set of variables, which will affect the runtime performance. However, there might be different ways to model RSP as an Integer Programming problem, e.g. the way we model the problem above is just one of them. Thus, we can improve the performance of Integer Programming approach by re-modeling the problem so that the number of integer variables can be reduced.

4.4. Schedule Generation Schemes based approach

4.4.1. Brute-force approach

As mentioned in section 3.3, SGS plays an important role in many heuristic approaches to solve the RCPSP. SGS is classified into serial and parallel based on the usage of time or activity incrementation. The basic idea of serial SGS is at each step of the process, an activity is selectively chosen from the set of ready-to-schedule activities. This approach is similar to the branch-and-bound technique that is described in section 3.2.2. Branch-and-bound has been used successfully to solve the Job Shop problem (section 3.1.2) as well as the RCPSP [13, 14].

With the concept of serial SGS, we propose a solution to solve the RSP by scheduling one activity at a time. Let's denote a timeslot, i.e. interval, as (*start, end*) and a time value *t* that is in this timeslot as $t \in (start, end)$. Then we our proposed approach can be formulated as:

```
Start:          Do Time-Table propagation of the original problem
                Create 2 sets of activities (events): $S_0 = \phi$ and $D_0 = \{E_i\}$, $i = 1,\ldots,n$
For i = 1 to n: Choose an event $E_i$ from $D_i$
                For each available timeslot ($start_j$, $end_j$) of $E_i$:
                    Assign each value t to the start time of $E_i$, $t \in (start_i, end_i)$
                    Update the available timeslots of $E_k \in D_i$ that shares at least 1
                    participant with $E_i$. If no available timeslot for $E_k$ then return.
                    Update $S_i = \{E_{i,}\}$, $D_i = D_{i-1} \cup \{E_i\}$
                    If $D_i = \phi$, save the makespan of the schedule
Return the minimized value of all makespans
```

We can see that the above algorithm is similar to the serial SGS method mentioned in section 3.3 because both of them are based on the activity incrementation approach. The main difference between the two algorithms is how we assign the start time of the current chosen activity. It is expected because unlike a RCPSP, the resource availability of our scheduling problem is not a constant during processing time and there is no activity precedence. Below are the steps demonstrating how we apply the above algorithm to the sample mentioned in section 4.1.

- Step 1: Time-Table propagation (details can be found in section 4.2.1)

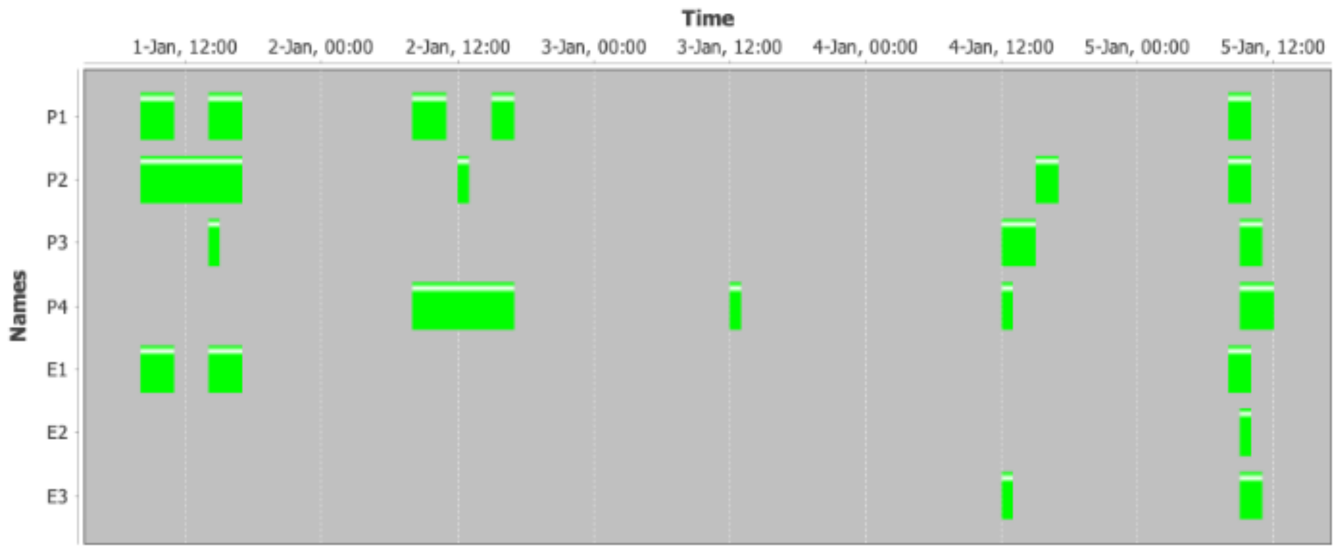

Figure 8. Time-Table propagation

- Step 2 (i = 1):
    - E1 is selected as the next activity without any heuristics. This selection can cause performance issue when the number of activities is large. A better solution combining heuristics and bounds that can lead to better performance is discussed in section 4.4.2.
    - From E1's list of possible timeslots, one is selected and the start time of E1 is set to this timeslot's start time. Figure 9 displays the result after this step so that E1's start time is set to 01/01 08:00. The period when E1 is scheduled is colored in blue. The corresponding periods of P1 and P2's available time are colored in blue as well to imply that during this time period, both P1 and P2 are busy attending event E1.

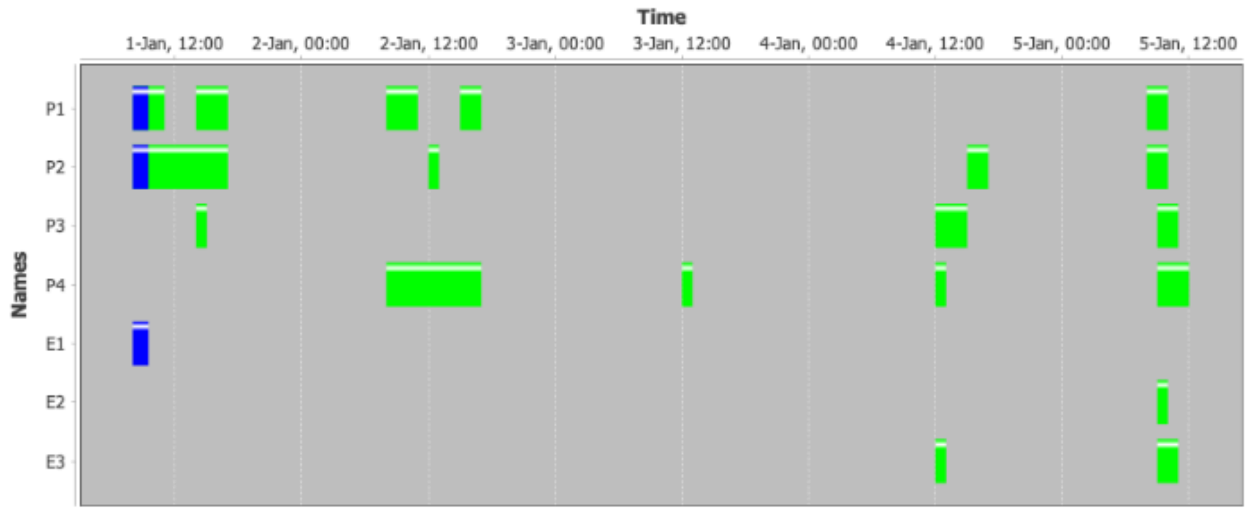

Figure 9. Brute force approach example – step 2

- Step 3 (i=2, E1 starts at 01/01 08:00):
    - Event E2 is chosen as the next activity. Again, E2 is selected randomly because there is no heuristic.
    - E2 is set to start at 01/05 09:00, which is the start time of its only possible timeslot. Figure 10 displays E2's start / end period and the corresponding available time of P1, P2, and P4 in orange color.

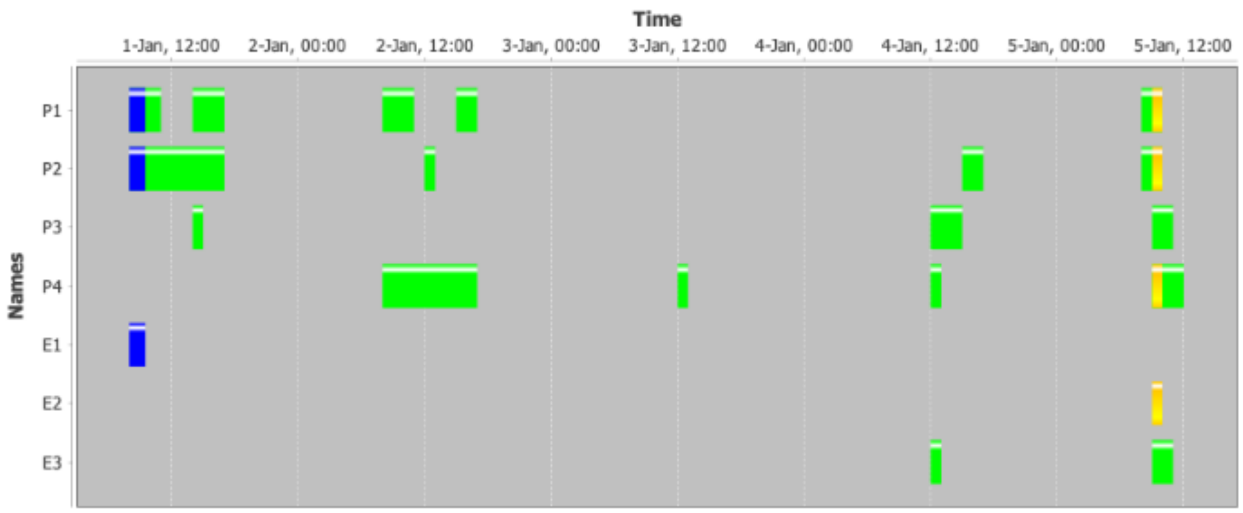

Figure 10. Brute force approach example – step 3

- Step 4 (i = 3, E1 starts at 01/01 08:00, E2 starts at 01/05 09:00):
    - There is only one event left, so E3 is chosen as the next activity.
    - After step 1, E3 has 2 possible timeslots starting from 01-04 12:00. So E3's start time is set to 01-04 12:00. E3's duration and its corresponding timeslot of P3, P4 are colored in magenta in Figure 11.

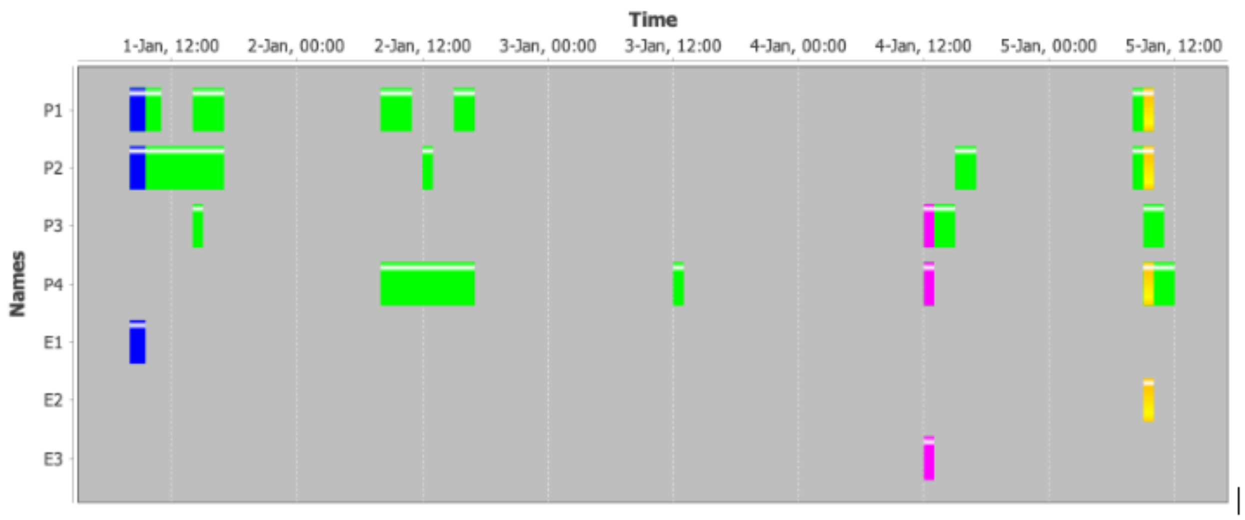

Figure 11. Brute force approach example – step 4

After step 4, we have a solution for the RSP that can satisfy all the constraints. However, the algorithm will not stop here. It will memorize the makespans of this solution, which is the end time of event E2, and then continue iterating through all the possible combinations of each event's valid start time. After each iteration, if a solution is found then the minimum makespan will be updated accordingly. For illustration purpose, we intentionally choose event E1, E2, E3 in order so that the current solution is optimal. In reality, it will take many iterations until an optimal solution is found, and the only way to prove a solution optimal is to go through all the iterations.

Obviously, the current algorithm is a brute-force approach because all possible combinations of activities' valid start time are considered. The complexity will be $O(k^n)$ where n is the number of activities and k is the number of available values that an event's start time can be assigned to. In section 4.4.2 we will discuss how to improve the current approach to get better performance. However, it is not clear if there is a polynomial complexity solution to solve RSP because the problem itself is NP-hard.

4.4.2. Improved approach

We observe that the ultimate goal of the algorithm is to minimize the makespan of all feasible solutions. During the iteration process, if an activity is scheduled so that its end time is greater than the current smallest makespan of all feasible solutions, then it is safe to discard the current iteration. This bounding technique can reduce a tremendous amount of unnecessary iterations compared to the brute-force approach.

Another observation is that currently the next activity chosen from the set D_i of the algorithm is selected randomly. If somehow, we can apply a "smarter" selection then the number of iterations may be reduced. In this report, we use these criteria to select the next activity in priority order as:

- Activity that has the less possibilities of start time (4.13)

- Activity that has the greatest number of other activities sharing as least one participant (4.14)
- Activity that has the longest duration (4.15)

Criterion (4.13) is used to limit the number of iterations as soon as possible. At a first glance, this criterion is not bringing much benefit because sooner or later we need to go through every activity's possible timeslots. However, our approach is a "depth-first" approach, therefore if we consider the whole problem as a tree and each activity as node, then the less branches the root has the better performance the algorithm can achieve. Obviously, the criterion (4.14) is used to remove as many possible values of the remaining activities' start time as possible. As mentioned in Section 4.2.1, once an activity is scheduled, we will update the availabilities of all related activities so this criterion will help reduce the search space as early as possible. Criterion (4.15) is used with the same purpose of the criterion (4.14) to reduce the search space as soon as possible.

At this stage, our original approach has been improved using the bounding and heuristic techniques. We observe that during the iteration, the same subproblems are considered again and again. More specifically, after assigning the start time for event E_i at each iteration, we remove that event from the current set of unscheduled events and update the available timeslots accordingly. In fact, this action is similar to creating a subproblem of the current considered problem. If the boundary of available timeslots is small while the number of activities is big, e.g. schedule 20 events within one week, then the chance of revisiting the same subproblem is high. This leads to a new promising improvement that if we can prove the newly created subproblem having the same solution with one of the already considered subproblems, then we do not need to consider that subproblem anymore. Unfortunately, there is no known algorithm with polynomial time complexity that can be used to check if two RSPs have the same solution. In our experiment, we apply a naïve comparison such that two scheduling problems are considered the same if:

- Each has the same number of events
- Each event E_{1i} in problem 1 must have a corresponding activity E_{2i} in problem 2 so that:
    - E_{1i}, E_{2i} have the same available timeslots
    - E_{1i}, E_{2i} have the same number of events that share at least one same participant
    - For each event in R_{1i}, there is an event in R_{2i} so that the two events have the same timeslots. R_{1i}, R_{2i} is the set of all related events of E_{1i} and E_{2i}, respectively

Obviously, the comparison above will not always return true even when the two problems have the same solution. But it will guarantee that once the returned result is true then the two problems are indeed having the same solution. Although the comparison is not ideal, it does give better performance when there are many activities scheduled in a short period of time.

4.5. Solution validation

To prove the correctness of the three approaches, we create a test program to verify if the solution can satisfy all constraints. The pseudo code of the program is described as follow:

```
For each event $E_i$:
    For each participant $P_j$ in $E_i$:
        If $P_j$ is unavailable during the period [start($E_i$), end($E_i$)]:      (4.16)
            Return False
        For each event $E_k$ that $P_j$ participates:                              (4.17)
            If $E_k$ and $E_i$ overlap:
                Return False
Return True
```

Code block (4.16) will verify if all related participants of an event are able to join the event during its period. Block (4.17) guarantees no overlap between two events that require at least one same participant.

The above program can verify if a solution satisfies all the constraints; however, it cannot guarantee that the solution is optimal. In the Constraint Programming and Integer Programming approaches, we define the objective function as the finish time of the latest event and dedicate it to the solver. Assuming the solver is correct, the solution returned by the solver should be optimal. In the brute-force approach, we iterate through every possibility of the next event from the list of ready-to-schedule events. This method will ensure that all cases are considered. Besides, only one solution that has the smallest makespan is kept during each iteration so the final solution should be optimal.

4.6. Benchmarks

As stated in the introduction section, RSP is considered as a NP-hard problem so we can expect that the runtime for each approach will increase immensely when the size of the problem is getting bigger. The goal of our benchmark is to give an overview of how well these proposed approaches can perform with different feasible and infeasible RSPs. Feasible RSP has an optimal solution while infeasible RSP does not have any solution that can satisfy all the constraints. Because there is no such direct research of RSP, we believe that these benchmarks are the first conducted on RSP.

Obviously, the size of a RSP depends on the number of events, the scatter of participant's availabilities and the number of participants of each event. Thus, we create different RSPs based on these criteria and monitor the runtime of each approach. For more accuracy, at each measuring step we create 50 RSPs randomly with the same criteria, then we use the average runtime that each approach performs on these RSPs as the benchmark indicator. The test machine has 16 GB RAM and 2.6 GHz Quad-Core Intel Core i7 CPU.

4.6.1. Benchmark based on number of events

In this experiment, we create RSPs with the fixed configuration as below:

Total participants	20
Available timeslot per participant	15
Number of participants per event	5
Project's span	2 weeks

Table 3. Input for number of events benchmark

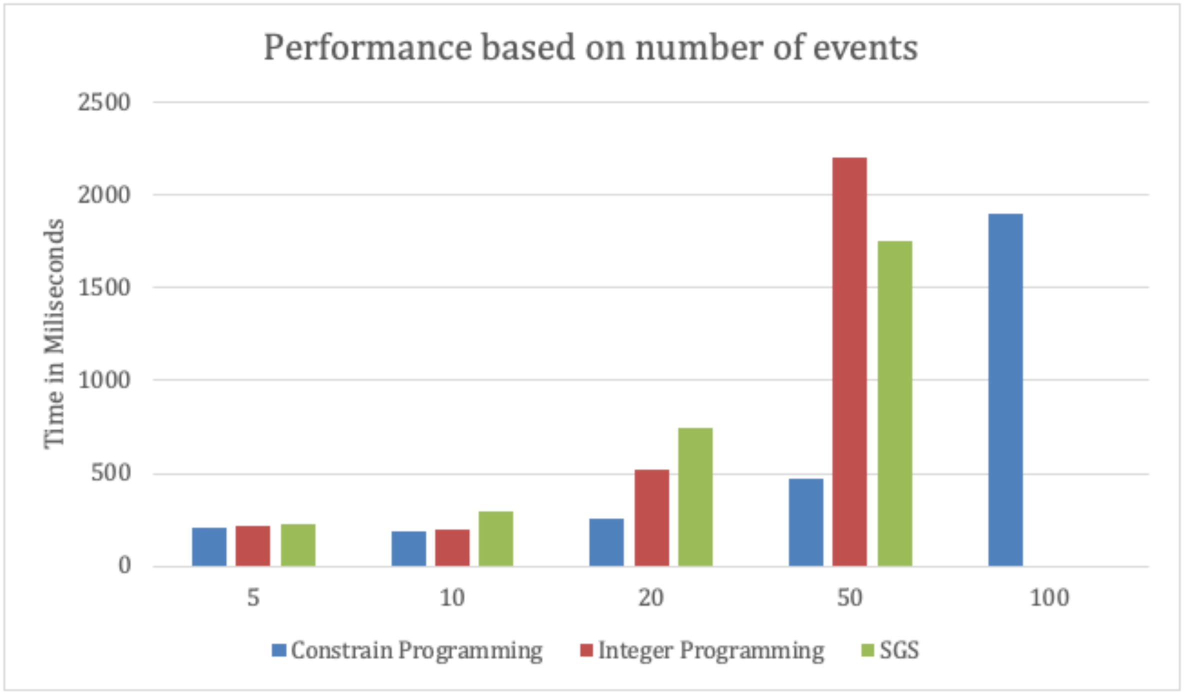

Figure 12. Benchmark based on number of events – Feasible

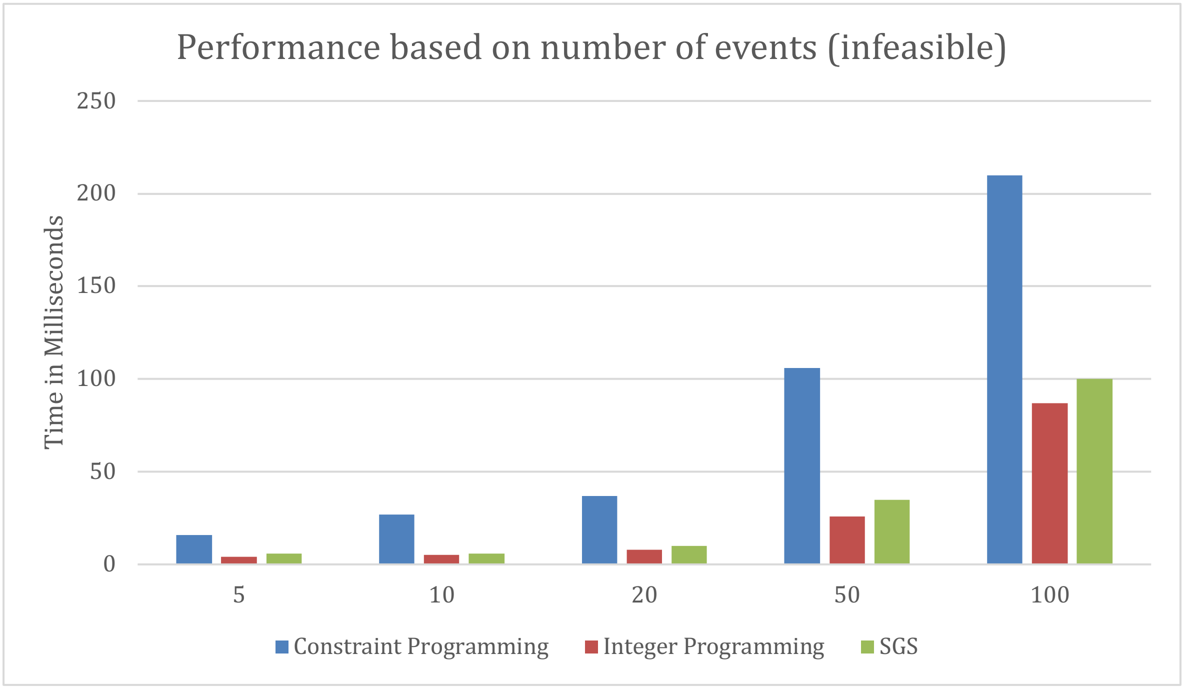

Figure 13. Benchmark based on number of events – Infeasible

As we can see in Figure 12, the Constraint Programing approach gives a superior performance compared to the other approaches for feasible problems. Especially, if the number of events is large (n = 100) then none of the Integer Programming or SGS approach can find an optimal solution in reasonable time, i.e. there is no optimal solution found within one hours. Figure 13 shows an interesting fact that for infeasible problem, Constraint Programming gives the worst performance.

4.6.2. Benchmark based on availability scatter

In this experiment, we increase the scatter of participants' availabilities to see how well the three approaches can perform. In theory, the more scattering of availabilities the more time the program needs to run. However, if the approach has a good bounding technique then there should not be many differences in performance.

Total events	20
Total participants	20
Available timeslot per participant	30
Number of participants per event	5

Table 4. Input for number of availabilities scatter benchmark

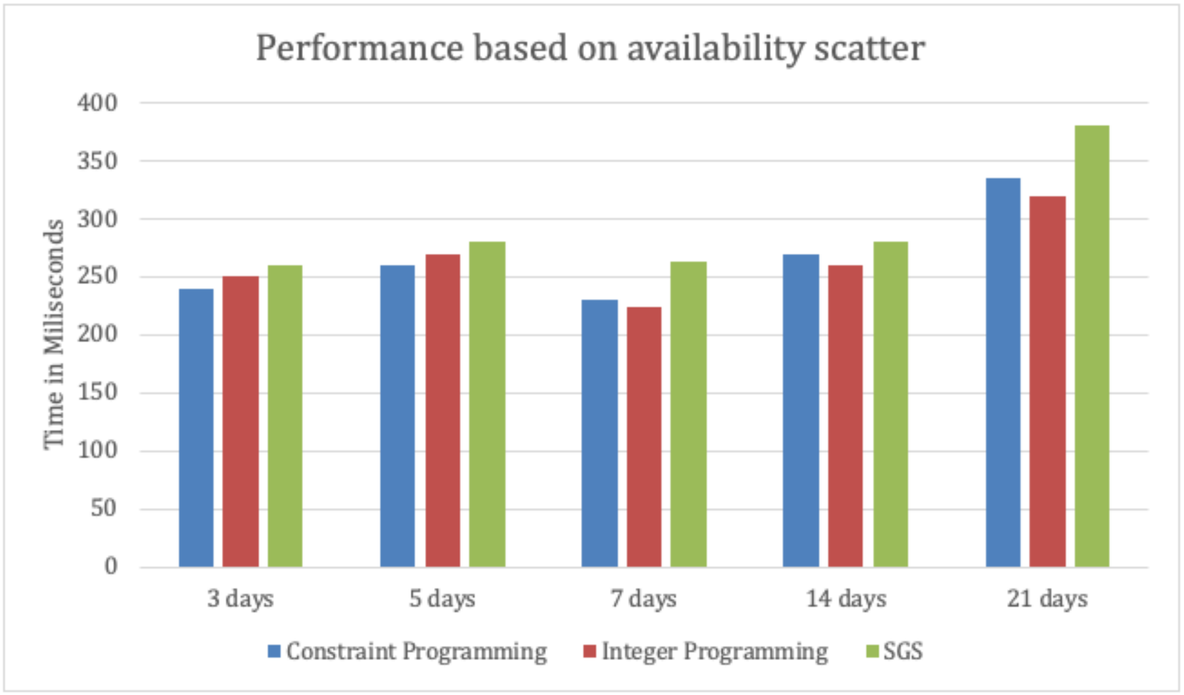

Figure 14. Benchmark based on availability scatter - Feasible

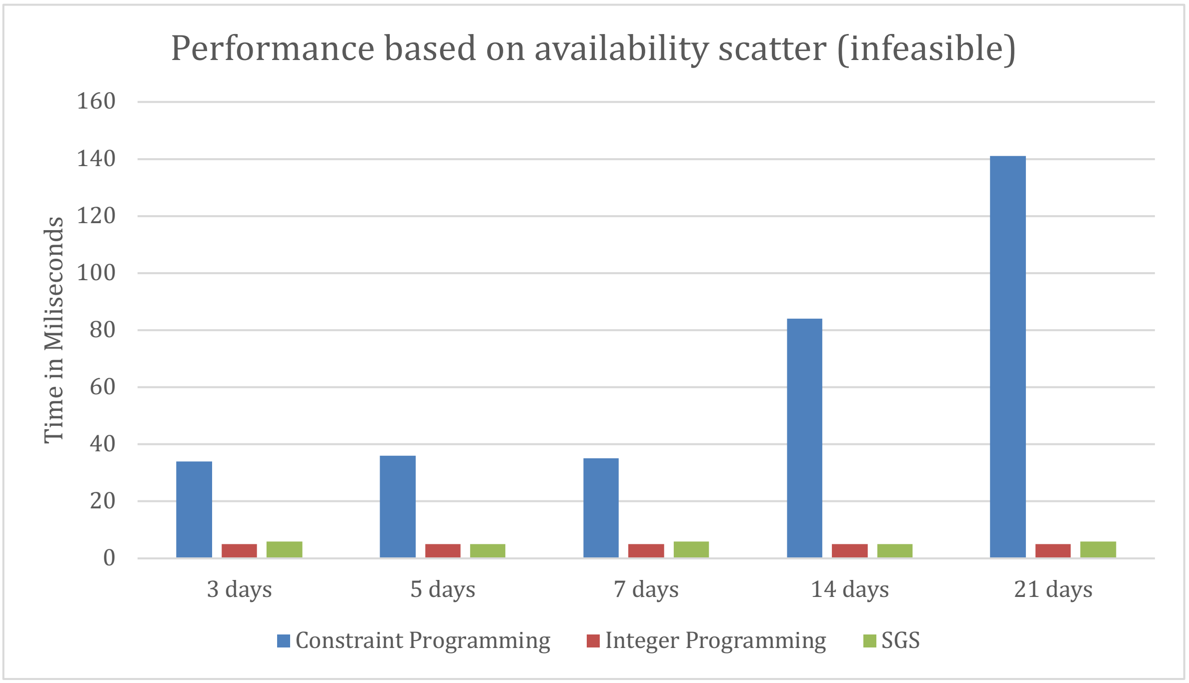

Figure 15. Performance based on availability scatter - Infeasible

We can see that even when the participant's availabilities expand over 21 days, there is not much difference in the computational time if the problem is feasible. This is understandable because all three approaches utilize the applied bounding technique. Surprisingly, Integer Programing gives a slightly better performance compared to Constraint Programing. For infeasible problem, Constraint Programming approach still performs worst. Integer Programming and SGS based approaches give a steady performance in term of proving a problem infeasible.

4.6.3. Benchmark based on number of participants per event

The last experiment is based on the average number of participants of each event. More participants mean more constraints are added; thus, the computational time will be increased. In order to get a feasible solution, we need to increase the number of timeslots as well as the project's span as below:

Total events	20
Total participants	20
Available timeslot per participant	40
Project's span	3 weeks

Table 5. Input for number of participants per event benchmark

The experiment gives an example of how well the Constraint Programming model can perform with feasible problem when the number of constraints is larger. Starting from the value of 5 participants per event, both Integer Programming and SGS approaches cannot give an optimal solution in reasonable time. If the number of participants per events is equal to 10 then even the Constraint Programming approach cannot perform well. In contrast, if the problem does not have any solution, then Integer Programming and SGS based approach have better performance.

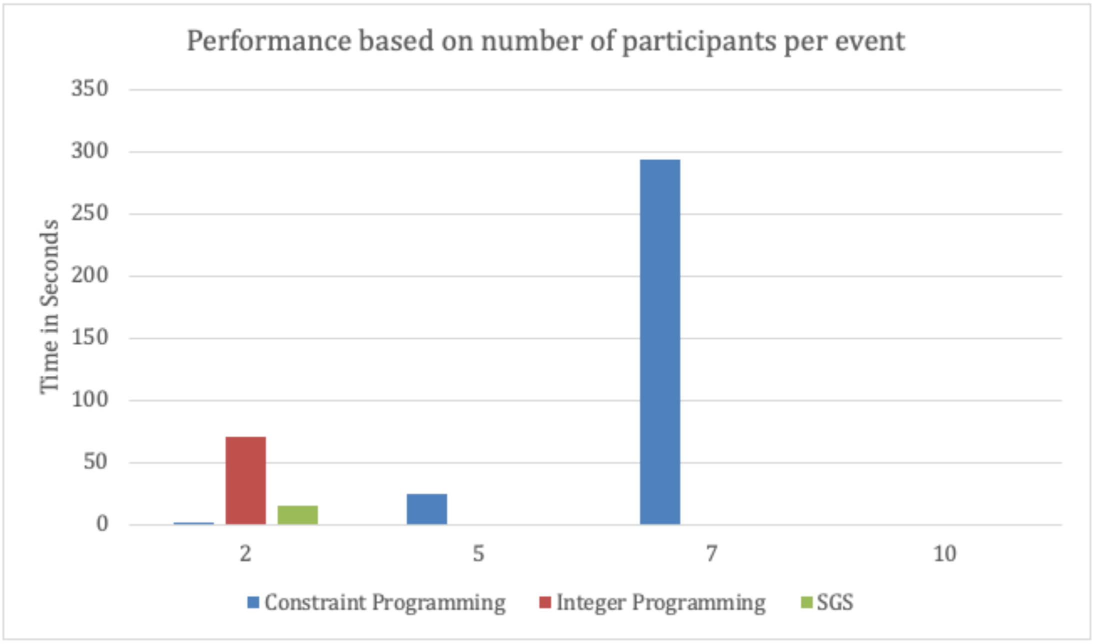

Figure 16. Benchmark based on number of participants per event - Feasible

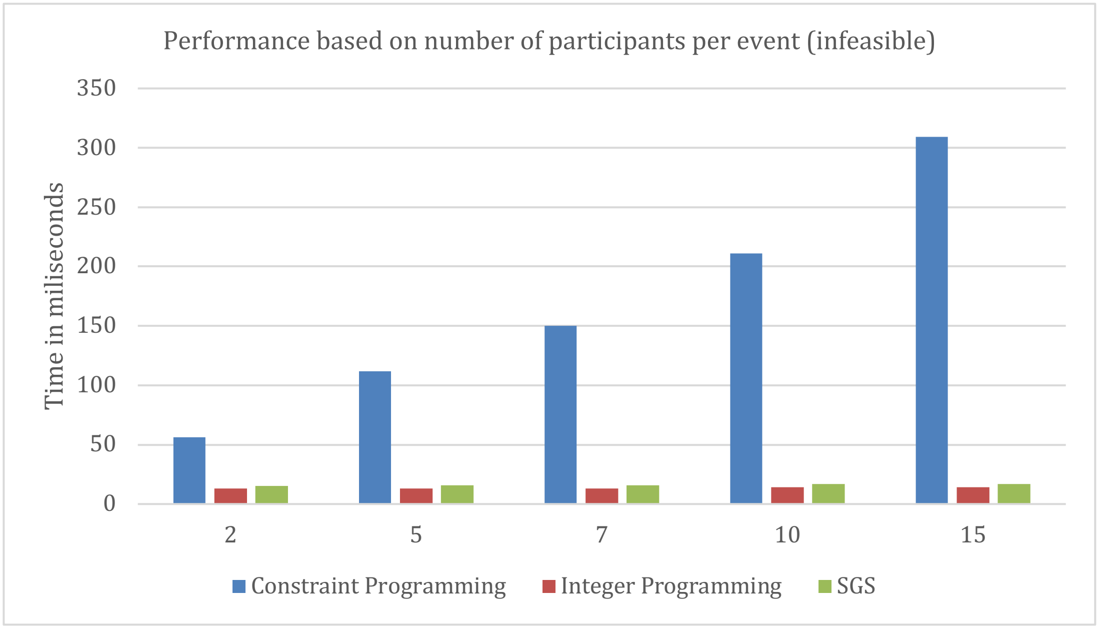

Figure 17. Performance based on number of participants per event - Infeasible

4.6.4. Summary

Table 6 quantifies the performance comparison, i.e. how many percentages faster an approach can perform compared to other approaches, between Constraint Programming (CP), Integer Programming (IP) and SGS. Negative value means the approach that is being compared to has better performance. "N/A" means the comparison is not possible because one of the two compared approaches cannot find a solution with reasonable time. Columns with grey background color is for performance comparison conducted on infeasible problems.

Criteria	Variant	CP/IP	CP/SGS	IP/SGS	IP/SGS	IP/CP	SGS/CP
Number of events	5	2.4	9.5	6.9	50	300	166.7
	10	-2.4	46.3	50	20	440	350
	20	101.9	169.2	33.3	25	362.5	270
	50	379	285.4	-19.6	34.6	307.7	202.9
	100	N/A	N/A	N/A	14.9	141.4	110
Availability scatter	3	4.1	8.3	4	20	580	466.7
	5	3.8	7.7	3.7	0	620	620
	7	-4.1	33.3	39.1	20	600	483.3
	14	-3.7	3.7	7.7	0	1580	1580
	21	-13	44.9	66.7	20	2720	2250
Number of participants per event	2	3400	525	-82.1	15.4	330.8	273.3
	5	N/A	N/A	N/A	23	761.5	600
	7	N/A	N/A	N/A	23.1	1053.8	837.5
	10	N/A	N/A	N/A	21.4	1407.1	1141.8
	15	N/A	N/A	N/A	21.4	2107.1	1717.6

Table 6. Quantified performance comparison in percentage

The benchmarks suggest that Constraint Programming approach outperforms the other approaches in term of running time when the problem is feasible. This is understandable because the characteristic of scheduling problem fits well with Constraint Programming methodology. However, if the RSP is large, e.g. when the number of participants per event is big enough, then even the Constraint Programming approach cannot solve the problem in reasonable time. This opens up the opportunity for further research on the RSP using techniques such as heuristic, local search and bounding, etc., which are successfully applied to solve the original RCPSP.

On the other hand, if the problem has no solution then all three approaches will give better running time performance overall. This observation can be explained that for infeasible problem, an inconsistency might be found after a few constraint propagations (in case of Constraint Programming or Integer Programming approaches) or after a few iterations (in case of SGS based approach). Especially, Integer Programming and SGS based approaches will give better performance than Constraint Programming approach. SGS based approach can perform well in this case because it follows "depth-first" paradigm with heuristic, which always chooses the "best" next activity that can find as many inconsistencies closer to the root as possible. This can help eliminate any infeasible iteration earlier. Unfortunately, we do not have a good explanation of why Integer Programming approach can give a very good performance compared to Constraint Programming approach when the problem is infeasible.

The above benchmarks are measured based on how fast an optimal solution, which satisfies all the time constraints and has the smallest makespan, can be found. There is possibility that the optimal solution found by each approach has the same makespan but "different" schedule. For example, if we change the start time of event E1 in Figure 6 to 01/01 14:00 then the makespan of the solution is not changed, thus remaining optimal. To this point, we might consider other factors of result quality that can help decide which optimal solution found by each approach is actually better. One promising factor is to minimize the number of consecutive events for each participant. However, it is not reasonable to judge the result of each approach based on consecutive events without adding this requirement to the problem.

If we really want to consider the factor of minimizing consecutive events, we can add it to the definition of the problem as a “soft” constraint, which can be violated if necessary. Then the goal of the original RSP is updated to find the solution with minimized makespan and minimized number of consecutive events. This modification of the original RSP with the additional soft constraint can be solved using the same approaches that we have discussed. Details of how to use soft constraint in scheduling problem can be found in [2], [15].

CONCLUSION

The rehearsal scheduling problem studied is similar to the original resource-constrained project scheduling problem. However, the rehearsal scheduling problem has different requirements such as no precedence constraints and resources' availabilities are not fixed during processing time. Thus, this problem requires different approaches to find the optimal solution. Three approaches are proposed in this report to solve rehearsal scheduling problem including Constraint Programming, Integer Programming, and Schedule Generation Schemes.

Among the three approaches, Constraint Programming has the superior performance. However, this benchmark does not mean that Integer Programming is not as powerful as Constraint Programming. The nature of rehearsal scheduling problem fits better with Constraint Programming paradigm, thus resulting in better performance. Flexibility is a characteristic of Integer Programming in problem modeling and it is proven through the Integer Programming approach.

We also introduce the brute-force approach based on Schedule Generation Schemes technique. Without heuristic and bounding, the brute-force approach's performance is worse than the Integer Programming approach's performance. The heuristic to choose next activity is an important factor to increase performance, thus additional research in this area could be done to further enhance performance. Although the dynamic programming implementation does not improve the overall performance tremendously, it gives us an opportunity to establish a promising improvement. We believe that by giving a better comparison between same subproblems, the algorithm can reduce the number of iterations and increase overall outcome.

REFERENCES

[1] R. Kolisch and S. Hartmann, "Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis," in *Project Scheduling. International Series in Operations Research & Management Scienc*, vol. 14, J. Węglarz, Ed., Boston, MA: Springer, 1999, pp. 147-178.

[2] P. Baptiste, C. L. Pape and W. Nuijten, Constraint-based scheduling : applying constraint programming to scheduling problems, vol. 39, Norwell, MA: Springer, 2001.

[3] K. R. Apt, Principles of Constraint Programming, New York, NY: Cambridge University Press, 2003.

[4] R. Dechter, Constraint Processing, San Francisco, CA: Morgan Kaufmann Publishers Inc., 2003.

[5] R. Barták, "Constraint-Based Scheduling: An Introduction for Newcomers," in *IFAC Proceedings Volume*.

[6] M. Fromherz, "Constraint-based scheduling," in *Proceedings of the 2001 American Control Conference. (Cat. No.01CH37148)*, Arlington, 2001.

[7] F. Rossi, P. v. Beek and T. Walsh, Handbook of Constraint Programming, New York, NY: Elsevier Science Inc., 2006.

[8] M. Conforti, G. Cornuejols and G. Zambelli, Integer Programming, vol. 271, Berlin: Springer International Publishing, 2014.

[9] M. L. Pinedo, Scheduling: Theory, Algorithms, and Systems, New York, NY: Springer-Verlag , 2012.

[10] C. L. Pape, "Constraint-based scheduling: principles and application," in *IEE Colloquium on Intelligent Planning and Scheduling Solutions*, London, 1996.

[11] "Google OR-Tools," Google, [Online]. Available: https://developers.google.com/optimization.

[12] L. Trilling, A. Guinet and D. L. Magny, "NURSE SCHEDULING USING INTEGER LINEAR PROGRAMMING AND CONSTRAINT PROGRAMMING," *IFAC Proceedings Volumes,* vol. 39, no. 3, pp. 671-676, 2006.

[13] P. Brucker, B. Jurisch and B. Sievers, "A branch and bound algorithm for the job-shop scheduling problem," *Special Volume Viewpoints on Optimization ,* vol. 49, no. 1-3, pp. 107-127, 1994.

[14] P. Brucker, S. Knust, A. Schoo and O. Thiele, "A branch and bound algorithm for the resource-constrained project scheduling problem," *European Journal of Operational Research,* vol. 107, no. 2, pp. 272-288, 1998.

[15] B. Ilham, J. A. Ferland and P. Michelon, "A multi-objective approach to nurse scheduling with both hard and soft constraints," *Socio-economic planning sciences,* vol. 30, pp. 183-193, 1996.

[16] C. Pakprajum and P. Jarumaneeroj, "An integrated MILP for music rehearsal problems with waiting time," *2017 4th International Conference on Industrial Engineering and Applications (ICIEA),* pp. 214-218, 2017.

[17] J. M. T. Goerlich and R. A.-V. Olaguíbel, "Heuristic algorithms for resource-constrained project scheduling: A review and an empirical analysis," in *Advances in Project Scheduling*, R. Słowiński and J. Węglarz, Eds., Elsevier, 1989, pp. 113-134.

[18] K. Moumene and J. A. Ferland, "Activity list representation for a generalization of the resource-constrained project scheduling problem," *European Journal of Operational Researc,* vol. 199, no. 1, pp. 46-54, 2009.

[19] S.-S. Liu and C.-J. Wang, "Optimizing project selection and scheduling problems with time-dependent resource constraints," in *Automation in Construction*, vol. 20, Elsevier, 2011, pp. 110-1119.

[20] G. Weil, K. Heus, P. Francois and M. Poujode, "Constraint programming for nurse scheduling," *IEEE Engineering in Medicine and Biology Magazine,* vol. 14, pp. 417-422, 1995.

[21] C. L. Pape, "Classification of scheduling problems and selection of corresponding constraint-based techniques," in *IEE Colloquium on Advanced Software Technologies for Scheduling*, 1993, pp. 1/1-1/3.

[22] B. M. Smith, "Constraint programming in practice: Scheduling a rehearsal," APES group, 2003.