An Instance Generator for the Multi-Skill Resource-Constrained Project Scheduling Problem

Bernardo F. Almeida[a]*, Isabel Correia[b], Francisco Saldanha-da-Gama[a]

[a] *Departamento de Estatística e Investigação Operacional / Centro de Matemática e Aplicações Fundamentais — Centro de Investigação Operacional, Faculdade de Ciências, Universidade Lisboa, 1749-016 Lisboa, Portugal*

[b] *Departamento de Matemática / Centro de Matemática e Aplicações, Faculdade de Ciências e Tecnologia, Universidade Nova Lisboa, 2829-516 Caparica, Portugal*

Abstract

Multi-skill Resource-Constrained Project Scheduling Problems consist of an extension of resource-constrained project scheduling problems in which resources are assumed to be flexible or, as they are more commonly referred to in the literature, multi-skilled. In many project scheduling problems that we find in the real world, resources can often master more than one skill and the activities require specific amounts of resources per each skill needed for their execution. This paper introduces an instance generator for the unit-capacity Multi-skill Resource-Constrained Project Scheduling Problem. The proposed procedure consists in generating a precedence network, a set of multi-skilled resources and a set of activities.

Keywords: Project Scheduling, Multi-skilled Resources, Instance Generator

*Corresponding author. *e-mail address*: bernardo.almeida@alunos.fc.ul.pt

1 Introduction

Multi-skill resource-constrained project scheduling problems define a class of challenging combinatorial optimization problems that have increasingly attracted the attention of the scientific community in the recent years. These problems extend the well-known Resource-Constrained Project Scheduling Problem (RCPSP) by assuming that resources can master more than one skill. Such resources are often referred to as flexible or multi-skilled. These specific extensions of the RCPSP are motivated by many practical applications (see, e.g., Bellenguez-Morineau and Néron 2007 and Correia et al. 2012).

In this paper, we focus on the multi-skill resource-constrained project scheduling problem (MSRCPSP) studied by Correia et al. (2012) which is a recent extension of the unit-capacity RCPSP. In this problem, each activity requires a predefined number of resources per each skill needed for its execution. Resources master one or several skills and can only be assigned to activities that require at least one of the skills they master. Resources are renewable, i.e., they can be assigned to multiple activities as long as their execution does not overlap in time. Each resource can contribute to an activity with at most one skill. When a resource is assigned to some activity it remains busy for the whole processing time of the activity. The objective in this problem is to determine the starting times of the activities as well as the set of resource/skill pairs that should be assigned to them in such a way that the makespan of the project is minimized. Two types of constraints are considered: precedence between the activities and resource constraints.

For the above problem, Correia et al. (2012) proposed a mixed-integer linear programming (MILP) formulation that was enhanced with valid inequalities. The same type of models was discussed in the more general context of project staffing problems by Correia and Saldanha-da-Gama (2015).

In order to test and evaluate the relevance of their contribution, Correia et al. (2012) generated a random set of instances since no benchmark instances of this problem existed at the time. Nevertheless, it is arguable whether that set is a representative set of instances of this problem. Despite the fact that defining "a representative set of instances" for some problem is still a matter of debate (see, i.e., Smith-Miles and Bowly 2015) it is nonetheless relevant to have tools for rationally generating sets of instances of a given problem in order to perform computational tests for evaluating new methodologies.

In this work, we propose an instance generator for the problem investigated by Correia

et al. (2012). This generator can be easily extended to other variants or extensions of the problem.

The remainder of this paper is organized as follows. Section 2 provides a detailed description of the specific problem whose instance generator we have developed. In Section 3, the instance generator is described. Section 4 discusses the set of instances created using the presented generator. In Section 5 we present our conclusions. The algorithms / pseudo codes associated with each phase of the instance generator are presented in the Appendix.

2 The Multi-Skill Resource-Constrained Project Scheduling Problem

In this section we formally describe the problem we are working with. Additionally, in order to help the reader to better capture all the features involved, we revisit the MILP model introduced by Correia et al. (2012).

2.1 Problem Description

We consider a project comprising a set of activities. In order to execute these activities, a finite set of renewable resources is available, each of which mastering one or several skills. These resources are homogeneous, i.e., for each skill, all the resources mastering that skill perform it with the same efficiency level. Furthermore, resources are unary (unit-capacity) in the sense that each resource can contribute with at most one skill it masters for only one activity which requires it at a time. Once a resource is assigned to perform a skill for some activity it remains busy for the whole processing time of the activity performing that same skill. Activities are linked by precedence relations, i.e., one activity can only start being processed after the completion of all its predecessors. Each activity has a fixed processing time and requires a certain number of resources for each skill necessary for executing it. The objective of the problem is to sequence and schedule all activities satisfying the precedence and resource constraints, such that the project makespan is minimized.

We consider the following notation:

Sets

$V = \{0, \ldots, i, \ldots, j, \ldots, n+1\}$	set of activities to be executed. Activities 0 and $n+1$ are dummy; they represent the beginning and the end of the whole project, respectively. Their processing time is equal to 0 and they do not have any skill/resource requirements.
$\mathcal{R} = \{1, \ldots, k, \ldots, K\}$	set of (renewable) resources.
$\mathcal{L} = \{1, \ldots, l, \ldots, L\}$	set of skills.
$\mathcal{L}_j$	set of skills required by activity $j \in V$.
$\mathcal{L}^k$	set of skills mastered by resource $k \in \mathcal{R}$.

Parameters

p_j	processing time of activity $j \in V$.
r_{jl}	number of "resource units" mastering skill $l \in \mathcal{L}_j$ required to process activity $j \in V$.

We assume that the values of p_j and r_{jl} are positive integers, $j \in V$, $l \in \mathcal{L}_j$.

Direct precedence relations can be represented by an acyclic activity-on-node network $G = (V, E)$. An arc $(i, j) \in E$ has a weight p_i and indicates that activity i is a (direct) predecessor of j. From this representation we can identify the following sets:

$Pred(j)$	set of immediate predecessors of activity j.
$\overline{Pred}(j)$	set of all predecessors of activity j (e.g., by transitivity).
$Succ(j)$	set of immediate successors of activity j.
$\overline{Succ}(j)$	set of all successors of activity j (e.g., by transitivity).

Based upon the previous notation the following sets can also be identified:

$\mathcal{R}_l = \{k \in \mathcal{R} : l \in \mathcal{L}^k\}$	set of resources mastering skill $l \in \mathcal{L}$.
$\mathcal{R}^j = \{k \in \mathcal{R} : \mathcal{L}_j \cap \mathcal{L}^k \neq \emptyset\}$	set of resources mastering at least one skill required to process activity $j \in V$.
$V^l = \{j \in V : l \in \mathcal{L}_j\}$	set of activities requiring skill $l \in \mathcal{L}$.
$V_k = \{i \in V : \mathcal{L}_i \cap \mathcal{L}^k \neq \emptyset\}$	set of activities requiring skills mastered by resource $k \in \mathcal{R}$.

Finally, we consider one additional set, denoted by A, representing all pairs of activities having no direct or transitive precedence relations[1] and that hence can be processed simultaneously if there are enough resources to meet their requirements simultaneously.

$$A = \{(i,j) \in V \times V \,:\, i \not\prec j \wedge j \not\prec i \wedge i \not\prec\!\!\prec j \;\wedge\; j \not\prec\!\!\prec i\}$$

2.2 MILP Model

Correia et al. (2012) proposed a mixed-integer linear programming model for the problem described above by making use of the following decision variables:

S_j : starting time for activity $j \in V$.

$$y_{ij} = \begin{cases} 1, & \text{if activity } i \text{ is completed before activity } j \text{ starts;} \\ 0, & \text{otherwise.} \end{cases} \qquad i,j \in V \wedge (i,j) \in A.$$

$$x_{jlk} = \begin{cases} 1, & \text{if resource } k \text{ contributes with skill } l \text{ for activity } j; \\ 0, & \text{otherwise.} \end{cases} \qquad j \in V,\, k \in \mathcal{R}^j,\, l \in \mathcal{L}^k \cap \mathcal{L}_j$$

The formulation proposed by Correia et al. (2012) is the following:

$$\text{minimize} \quad S_{n+1} \tag{1}$$

$$\text{subject to:} \quad S_j \geq S_i + p_i \qquad i,j \in V \wedge (i,j) \in E, \tag{2}$$

$$S_j \geq S_i + p_i - M\,(1 - y_{ij}) \qquad i,j \in V \wedge (i,j) \in A, \tag{3}$$

$$y_{ij} + y_{ji} \leq 1 \qquad i,j \in V \wedge (i,j) \in A, \tag{4}$$

$$\sum_{k \in \mathcal{R}_l} x_{jlk} = r_{jl} \qquad j \in V,\, l \in \mathcal{L}_j, \tag{5}$$

$$\sum_{l \in \mathcal{L}^k \cap \mathcal{L}_j} x_{jlk} \leq 1 \qquad k \in \mathcal{R},\, j \in V_k, \tag{6}$$

$$\sum_{l \in \mathcal{L}^k \cap \mathcal{L}_i} x_{ilk} + \sum_{l \in \mathcal{L}^k \cap \mathcal{L}_j} x_{jlk} \leq y_{ij} + y_{ji} + 1, \qquad i,j \in V,\, k \in \mathcal{R}^i \cup \mathcal{R}^j,\; i,j \in V_k \wedge (i,j) \in A, \tag{7}$$

[1] A direct (transitive) precedence relation is represented by $i \prec j$ and $(i \prec\!\!\prec j)$ if i is a direct (transitive) predecessor of j.

$$S_j \geq 0 \qquad j \in V, \tag{8}$$

$$x_{jlk} \in \{0,1\} \qquad k \in \mathcal{R},\ j \in V_k,\ l \in \mathcal{L}^k \cap \mathcal{L}_j, \tag{9}$$

$$y_{ij} \in \{0,1\} \qquad i,j \in V \wedge (i,j) \in A. \tag{10}$$

The objective function (1) represents the starting time of dummy activity $n+1$, which is equal to the project makespan (to be minimized). Constraints (2) ensure the precedence relations. Constraints (3) determine the values for y_{ij} for each pair of activities $(i,j) \in A$ (not having any type of precedence relations). Constraints (4) complement constraints (3) by stating that for each pair $(i,j) \in A$: i and j are processed simultaneously, i starts after j is completed or j starts after i is completed. Constraints (5) ensure that the skill requirements of the activities are fulfilled through the assignment of the appropriate number of resources. Constraints (6) ensure that each resource contributes with at most one skill (that it masters) to an activity. Constraints (7) limit the assignment of each resource to at most one activity at a time. Note that these constraints need only to be considered for the pairs of activities $(i,j) \in A$. Constraints (8)–(10) are the domain constraints.

3 Instance Generator

In this section we develop an instance generator for the problem described in Section 2. The proposed methodology is somehow inspired by the work developed by Kolisch and Sprecher (1996) for the RCPSP.

The generator we propose consists of three components:

1. precedence graph generation (Section 3.1);
2. multi-skill resource generation (Section 3.2);
3. activity generation (Section 3.3).

3.1 Generation of a Precedence Network

As it was already mentioned above, in a project scheduling problem, the time-dependence between activities can be represented by a precedence network which is an acyclic Activity-On-Node (AON) graph $G = (V, E)$ where V is the set of activities and E is the set of arcs. We assume that $V = \{0, 1, \ldots, n, n+1\}$ where, as previously introduced, n is the

number of activities to be executed and 0 and $n+1$ are dummy activities. Each arc in G represents a precedence relation between the two activities it connects. It is assumed that $i < j, (i,j) \in E$. A precedence network can be obtained by generating precedence relations between pairs of activities. A project scheduling precedence network is said to be feasible only if its integrating arcs are non-redundant. An arc (i,j) is said to be redundant if it establishes a precedence which results by transitivity by at least two direct precedences already in the network. For instance if activity i precedes j and j precedes u, the insertion of the arc (i,u) in the network would introduce redundancy as there is already a transitive precedence, $i \prec\prec u$ assured by the arcs (i,j) and (j,u).

In order to generate a precedence network, some input data has to be provided. In particular we consider the following:

$n = \|V \setminus \{0, n+1\}\|$	number of activities in the project.
$nStart$	number of starting activities, i.e., activities having no predecessors.
$nFinish$	number of concluding activities, i.e., activities having no successors.
$MaxPred$	maximum number of predecessors for each activity.
$MaxSucc$	maximum number of successors for each activity.
NC	network complexity.

When generating a precedence network we must ensure that apart from the initial dummy activity (representing the beginning of the project), all the other activities have at least one predecessor. Furthermore, apart from the final dummy activity (representing the conclusion of the whole project), all activities must have at least one successor. The previous conditions ensure that the precedence network is connected in the sense that for each $j \in V$, there is at least one path connecting the dummy node 0 with j and at least one path connecting j with the dummy node $n+1$.

One measure that is often considered when generating precedence networks is the Network Complexity (NC) that can be formally defined as the average number of non-redundant arcs in the graph. In the scope of project scheduling, NC is the average number of direct successors of the activities $j \in V$.

Giving the input above presented, we propose a procedure for generating a precedence network that can be summarized as follows:

Procedure for generating a precedence network.	
Step 0:	(Initialization) Set nodes $\{1, \ldots, nStart\}$ to represent the starting activities. Set nodes $\{n-nFinish+1, \ldots, n\}$ to represent the concluding activities. Define the arcs $(0, j)$, $j \in \{1, \ldots, nStart\}$. Define the arcs $(j, n+1)$, $j \in \{n - nFinish + 1, \ldots, n\}$.
Step 1:	Assign one predecessor to each activity having no predecessors (apart from activity 0 and starting activities).
Step 2:	Assign one successor to each activity having no successors (apart from activity $n+1$ and concluding activities).
Step 3:	Add/remove arcs until the desired value for NC is achieved.

We discuss now the steps itemized in the above procedure.

Step 0 (Initialization).

This step consists of two phases: in the first one we set the first $nStart$ nodes (apart from 0) as the starting activities and the last $nFinish$ nodes (excluding $n+1$) as the concluding activities.

In the second phase, arcs are created for connecting the dummy 0 to every starting activity and also for connecting the concluding activities to the dummy $n+1$ (see Figure 1). At this stage, the number of non-redundant arcs in the graph equal to the sum of nStart and nFinish.

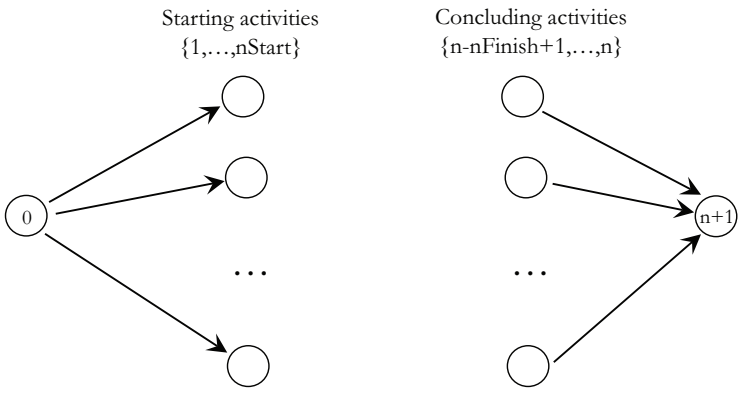

Figure 1: Network after the initialization step

Step 1.

This step is performed for each activity $j \in \{nStart + 1, \ldots, n\}$ (since activities $\{1, \ldots, nStart\}$ have already the dummy 0 as a predecessor) as follows:

(i) randomly[2] select one activity $i \in \{1, \ldots, j-1\}$ such that $|Succ(i)| < MaxSucc$. Note that if j is a concluding activity, i has to be randomly selected in the set $\{1, \ldots, n - nFinish\}$ to avoid redundancy in the set of concluding activities.

(ii) create the arc (i, j) and increment by one unit the number of non-redundant arcs in the network.

This step is formalized in Algorithm 1, presented in the Appendix.

Step 2.

This step excludes activities $\{n - nFinish + 1, \ldots, n\}$ since all these are concluding activities and cannot have other successors than the dummy $n + 1$.

We start with the highest numbered activity (without successors) and proceed backwards. Denote by j one such activity. We propose the following methodology:

(i) randomly select one activity $u \in \{j + 1, \ldots, n\}$ such that $|Pred(u)| < MaxPred$ and $u \notin Succ(i)$ for $i \in \overline{Pred}(j)$. Note that if j is a starting activity, u has to be randomly selected in the set $\{nStart + 1, \ldots, n\}$ to avoid redundancy in the set of starting activities.

(ii) define the new arc (j, u) and increment by one unit the number of non-redundant arcs in the network.

We note that in (i) we are avoiding redundancy in the network because if an arc (j, u) is to be considered and if u is already an immediate successor of $i \in \overline{Pred}(j)$, then arc (i, u) would become redundant.

It is also important to point out that in this step we are not following the procedure proposed by Kolisch and Sprecher (1996), which starts assigning successors to the lowest numbered activities. The reason is that using the methodology proposed by Kolisch and Sprecher (1996) a connected precedence network may never be obtained; thus the procedure may get stuck and requiring to be restarted. The latter is due to

[2]all random generated/selected data are pseudo-random numbers with discrete uniform distribution.

the fact that an upper bound on the maximum predecessors per activity, $MaxPred$, is imposed. Next, we illustrate this situation.

Let $V = \{0, 1, 2, 3, 4, 5, 6\}$. Then, we have 5 activities to process (0 and 6 are dummies). Assume now that $MaxPred = 2$, $nStart = 1$ and $nFinish = 1$. This means that activity 1 will be the only starting activity, activity 5 will be the only concluding activity and thus the other activities $2, 3, 4$ must succeed (precede) 1 (5).

Defining node 5 as a successor of both nodes 1 and 2 will never lead to a connected network because node 5 is the only feasible successor to activity 4 but has already reached its maximum number of predecessors $|Pred(5)| = MaxPred = 2$.

In fact, since activity 4 is an intermediate activity, it cannot be connected to the dummy sink 6. Accordingly, the resulting network would not be connected and the process would have to start from the beginning as suggested by Kolisch and Sprecher (1996).

Our proposal for Step 2 is formalized in the Algorithm 2, presented in the Appendix.

Step 3.

This step aims at adding (non-redundant) arcs to the network (thus establishing new precedences) such that the desired value for the Network Complexity (NC) is obtained. If the current Network Complexity (NC) is smaller than the desired one we add new arcs into the network by repeating the following steps:

1. randomly select two activities (nodes) i and j with $i < j$ such that $(i, j) \notin E$.
2. if the new arc (i, j) is not redundant then insert it into E and increment the number of non-redundant arcs in the network.

For the selected nodes i and j, $i < j$, Kolisch and Sprecher (1996) identify four types of redundancy that can emerge if the arc (i, j) is created:

(i) $j \in \overline{Succ(i)}$

(ii) $\exists\ u \in \overline{Succ(j)} : Pred(u) \cap \overline{Pred(i)} \neq \emptyset$

(iii) $Pred(j) \cap \overline{Pred(i)} \neq \emptyset$

(iv) $Succ(i) \cap \overline{Succ(j)} \neq \emptyset$

In addition to these types of redundancy we must also consider the cases referred in Steps 1 and 2, which are related respectively to having i and j both in $\{n - nFinish + 1, \ldots, n\}$ or both belonging to the set $\{1, \ldots, nStart\}$. In any of these cases, the generated pair is again ignored since no precedence can exist between activities inside those sets (otherwise we would have redundancy).

If the existing Network Complexity is higher than the desired we repeat the following step until we get the desired value:

1. select an arc $(i, j) \in E$ that can be removed from the network and remove it.

An existing arc (i, j) can be removed from the network as long as the network remains connected. Accordingly, an arc (i, j) can be removed only if node i keeps having at least one successor and node j remains with at least one predecessor.

We note that, again, in this step 3, we are not following exactly the procedure (Step 4) proposed by Kolisch and Sprecher (1996). That procedure only considers adding arcs to the network. In our case, we also consider removing them.

The Step 3 for generating a precedence network is formalized in Algorithm 3, presented in the Appendix.

3.2 Generation of Resources

In order to obtain a MSRCPSP instance we need more than the precedence network generated in the previous section; we need also to generate a set of multi-skilled resources. This is the goal of the second phase of our scheme. This phase requires the following input:

$K = \|\mathcal{R}\|$	total number of resources to generate.
$maxSkills$	maximum number of skills a resource can master. Obviously, it should be $1 \leq maxSkills \leq L$.

Each resource is fully defined by the skills it masters. For each resource $k \in \mathcal{R}$ we propose generating the corresponding information as follows:

1. randomly select $|\mathcal{L}^k|$ in the set $\{1, \ldots, maxSkills\}$;
2. randomly select $|\mathcal{L}^k|$ different skills in $\mathcal{L}$, thus obtaining $\mathcal{L}^k$.

After generating the skills mastered by the resources, it is necessary to check if all skills are mastered by at least one resource. If this is not the case, the above procedure is restarted.

3.3 Generation of Activities

In the MSRCPSP, each activity $j \in V$ is defined by three components: (i) a processing time (p_j), (ii) a set of required skills ($\mathcal{L}_j$), and (iii) a number of resources mastering each required skill (r_{jl}, $l \in \mathcal{L}_j$).

We regulate the process of generating the second and third components above by making use of two measures considered by Correia et al. (2012): the Skill Factor and the Modified Resource Strength. For each activity $j \in V \setminus \{0, n+1\}$, the skill factor ($SF_j \in]0, 1]$) measures the proportion of the total skills each activity requires. In this work we consider: (i) instances where all activities have the same skill factor and hence $SF_j = SF, j \in V \setminus \{0, n+1\}$, and (ii) instances where activities can have different skill factors. In this case the number of skills each activity requires is randomly generated. A skill factor of 1 for some activity means that the activity requires at least one unit of all skills.

The Modified Resource Strength ($MRS \in]0, \infty[$) measures how demanding the activities are in terms of the available resources. For a given instance, this measure is computed as the ratio between the number of existing resources and the total number of resources assigned to all activities:

$$MRS = \frac{K}{\sum_{j \in V} \sum_{l \in \mathcal{L}_j} r_{jl}}$$

Higher values of MRS yield instances with more available resources or less demanding activities which, in any case, typically, leads to "easier" instances.

The MRS is an input parameter in the process of generating instances to the MSRCPSP; it makes possible to randomly generate activity skill requirements by calculating the total demand of all activities.

We would like to point out that the skill factor and the modified resource strength described above are adapted from the resource factor and resource strength, respectively, that are usually considered for the classical RCPSP (cf. Kolisch and Sprecher (1996)).

Nevertheless, the MRS differs from the original resource strength proposed by Kolisch and Sprecher (1996) for the RCPSP in two ways: by considering resource aggregation

in opposition to Kolisch and Sprecher (1996) who consider RS to be resource specific, and by taking into account total demand (aggregation over time), where Kolisch and Sprecher (1996) take into account peak demands (e.g., when a Earliest Start Scheduled is considered).

The generation of the information concerning the activities in a MSRCPSP requires the following input:

$minProcTime$	minimum processing time.
$maxProcTime$	maximum processing time.
SF	skill factor.
	This value can be the same for all activities or different (randomly generated for each activity).
MRS	modified resource strength.
$maxResAct$	maximum number of resources per activity.
$maxResSkill$	maximum number of resources per each needed skill.

Giving this input, we propose a procedure for generating the information for the activities, which can be summarized as follows:

Procedure for generating the information concerning the activities.	
Step 1:	Processing time generation.
Step 2:	Definition of the sets $\mathcal{L}_j$.
Step 3:	Obtain the desired MRS.

We discuss now these steps.

Step 1.

This step is accomplished as follows:

1. set $p_0 = p_{n+1} = 0$ (the dummy activities have null processing time);
2. for each activity $j \in \{1, \ldots, n\}$, randomly generate its processing time, p_j within the set $\{minProcTime, \ldots, maxProcTime\}$.

Step 2.

The definition of the specific skills each activity requires is done as follows:

1. the dummy activities have no skill requirements, i.e., $|\mathcal{L}_0| = |\mathcal{L}_{n+1}| = 0$
2. for each activity $j \in \{1, \ldots, n\}$ we randomly select $|\mathcal{L}_j|$ different skills for defining the set $\mathcal{L}_j$. For a given $l \in \mathcal{L}$ that selection is made by setting the respective $r_{jl} = 1$.

   In case all activities require the same number of skills, $|\mathcal{L}_j|$ is always equal to $\lceil L \times SF \rceil$; otherwise, it should vary.

Since all the available skills are mastered by at least one resource, setting $r_{jl} = 1, l \in \mathcal{L}_j$ is always feasible. Similarly to what we discussed for the resources, it is mandatory that every skill $l \in \mathcal{L}$ is demanded by at least one activity.

We denote by ρ the number of already assigned resources. This value is initialized to 0, and it is incremented as resources are added to the requirements of the activities. In the end of this step $\rho = \sum_{j \in V} |\mathcal{L}_j|$.

This step is formalized in Algorithm 5, presented in the Appendix.

Step 3.

After selecting the skills required by each activity, we can now focus on meeting the MRS by incrementing the skill requirements of the activities. We start by computing the total number of resources, which have to be assigned to all activities, in order to meet the MRS as follows:

$$\text{total number of resources required by all activities} = \left\lfloor \frac{K}{MRS} \right\rfloor$$

Afterwards, we focus on incrementing the skill requirements of the activities until the sum of their requirements equals the total number of resources required by all activities. It is essential to respect the upper bounds on both the maximum number of resources per activity ($maxResAct$) and the maximum number of resources per each skill required ($maxResSkill$). In every instance generated by this procedure, the minimum number of resources required per each non-dummy activity is equal to $|\mathcal{L}_j|$, defined in Step 2, and the minimum number of resources per each skill required is assumed to be 1.

Step 3 is illustrated below:

1. while the number of already assigned resources, ρ, is less than the total number of resources required by all activities, randomly select an activity $j \in \{1, \ldots, n\}$ and a random skill $l \in \mathcal{L}_j$, then increment the corresponding value r_{jl} by one unit if $r_{jl} < maxResSkill$ and $\sum_{l' \in \mathcal{L}_j} r_{jl'} < maxResAct$.
2. check whether the skill requirements of activity j can be met. If so, then ρ is incremented by one unit; otherwise r_{jl} is decremented by one unit.

Checking whether the skill requirements of an activity $j \in \{1, \ldots, n\}$ are feasible (in the sense that we have enough resources to perform the activity with those requirements) can be easily done by solving a feasibility problem. The idea is to check whether a feasible flow exists in a specific network. For some activity $j \in \{1, \ldots, n\}$ the corresponding network is a bipartite graph built as follows:

- The set of nodes is defined by:

  - a source node s;

  - a set of nodes $\mathcal{R}_j$ associated with the resources that master at least one skill required by activity j;

  - a set of nodes $\mathcal{L}_j$ associated with the skills required to execute activity j;

  - a sink node t.

- The set of arcs contains:

  - a set of arcs (s, k), $k \in \mathcal{R}_j$ with minimum throughput 0, capacity 1 and cost 0;

  - a set of arcs (k, l), $k \in \mathcal{R}_j, l \in (\mathcal{L}^k \cap \mathcal{L}_j)$ with minimum throughput 0, capacity 1 and cost 0;

  - a set of arcs (l, t), $l \in \mathcal{L}_j$ with minimum throughput and capacity equal to the current value of r_{jl} and cost 0;

The arcs exiting the source node s and the intermediate arcs have capacity 1 to ensure that each resource is selected at most once and thus can contribute with at most one skill to the execution of the activity. The arcs arriving at the sink node t have minimum throughput and capacity equal to the total number of resources necessary to perform that skill, r_{jl}, which vary in course of the algorithm.

If a feasible flow can be found in the above network then then we know that there are enough resources to meet the requirements of all skills required to process activity j.

Step 3 is formalized in Algorithm 6, presented in the Appendix.

4 A New Set of Benchmark Instances

In this section we introduce larger instances for the MSRCPSP which were generated using the methodology presented in Section 3. We note that a set of larger instances for the MSRCPSP was missing in the literature.

The input parameters which are used to create this new set of instances are:

- 40 activities with processing times in the set $\{1, \ldots, 10\}$;
- 4 skills;
- $NC \in \{1.5, 1.8, 2.1\}$;
- $SF \in \{0.5, 0.75, 1, \text{variable}\}$ The word "variable" means that for each activity the number of skills was randomly generated in the set {2, 3, 4}.
- the MRS, the SF and the corresponding number of resources are presented in Table 1;
- each activity requires at most 7 resources for each needed skill and no upper bound on the number resources per activity is imposed;
- each resource masters 1, 2 or 3 skills among the 4 available;
- the number of resources in each instance varies from 20 to 60 depending on SF and MRS.

SF =1		SF = 0.75		SF = 0.5	
MRS	$\lvert\mathcal{R}\rvert$	MRS	$\lvert\mathcal{R}\rvert$	MRS	$\lvert\mathcal{R}\rvert$
0.0625	40	0.0625	30	0.0625	20
0.078125	50	0.079167	38	0.078125	25
0.09375	60	0.09375	45	0.09375	30

Table 1: Modified Resource Strength and number of resources for the new set of instances.

For each combination of SF, NC and MRS, 5 instances were generated which yields a total of 180 instances.

Below, we present an example of input parameters used to create a subset of instances as well as an example of one of those instances.

Input	
General Parameters:	n: 40
	K: 20
	L: 4
Precedence Network:	$nStart$, $nFinish$: 3
	$MaxPred$, $MaxSucc$: 3
	NC: 1.5
Resources:	$maxSkills$: 3
Activities:	$MinProcTime$: 1
	$MaxProcTime$: 10
	SF: 0.5
	MRS: 0.0625
	$maxResSkill$: 7
	$maxResAct$: K

The output from the instance generator is presented below.

The sets $Succ(j), j \in V \setminus \{0, n+1\}$, are used to define the precedence network. For each non-dummy activity j, we consider a boolean array with size n to represent $Succ(j)$. Each position of that array is associated with one activity u, hence a "1" in that position means that u is a direct successor of j.

Resources are fully defined by the skills they master. The definition of each activity j comprehends its processing time (p_j) and the number of resources required by each skill needed for their execution: $r_{jl} : l \in \mathcal{L}_j$.

5 Conclusions

In this technical report we present an instance generator for the unit-capacity MSRCPSP which can be easily adapted to solve extensions of the referred problem. The motivation for developing such generator arises from the need of evaluating and comparing the performance of exact and heuristic methods for the problem. Moreover the goal was to obtain

j	$Succ(j)$
1	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
2	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
3	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4	0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
...	
35	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
36	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
37	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
38	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
39	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
40	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Figure 2: Partial Precedence Network.

k	$l=1$	$l=2$	$l=3$	$l=4$
1	1	0	0	1
2	0	1	1	1
3	0	1	1	1
4	0	1	1	1
5	1	0	1	1
...				
15	0	1	0	0
16	0	0	1	0
17	0	0	0	1
18	1	0	1	1
19	0	1	0	0
20	1	1	1	0

Figure 3: Resources.

j	p_j	r_{j1}	r_{j2}	r_{j3}	r_{j4}
1	10	0	3	0	4
2	9	3	3	0	0
3	2	0	0	3	5
4	8	4	0	0	3
5	9	3	0	3	0
...					
35	1	5	0	0	4
36	5	0	4	0	2
37	4	0	5	2	0
38	4	0	0	4	6
39	9	0	5	3	0
40	8	0	5	0	3

Figure 4: Activities.

a comprehensive set of benchmark instances for the problem consisting of the instances created by Correia et al. (2012) together with a new set of larger instances.

References

Bellenguez-Morineau, O. and Néron, E. (2007). A Branch-and-Bound Method for Solving Multi-Skill Project Scheduling Problem. *RAIRO Operations Research*, 41:155–170.

Correia, I., Lourenço, L. L., and Saldanha-da-Gama, F. (2012). Project scheduling with flexible resources: formulation and inequalities. *OR Spectrum*, 34:635–663.

Correia, I. and Saldanha-da-Gama, F. (2015). A modeling framework for project staffing and scheduling problems. In Schwindt, C. and Zimmermann, J., editors, *Handbook on Project Management and Scheduling*, volume 1, pages 547–564. Springer. Switzerland.

Kolisch, R. and Sprecher, A. (1996). PSPLIB - A project scheduling problem library. *European Journal of Operational Research*, 96(1):205 – 216.

Smith-Miles, K. and Bowly, S. (2015). Generating new test instances by evolving in instance space. *Computers & Operations Research*, 63:102–113.

Apendix: Pseudo Codes

In this appendix we present the pseudo codes for the whole instance generator. In particular, we formalize 5 algorithms properly introduced in Section 3.

Algorithm 1: Generating a precedence network — Step 1.

Data: $nStart$, $nFinish$, $MaxSucc$, $Pred(j) : j \in V$, $Succ(j) : j \in V$

Result: Randomly selects a predecessor i for each non-dummy activity $j \in V : Pred(j) = \emptyset$

$nonredarcs \longleftarrow nStart + nFinish$; // number of non-redundant arcs in the graph;

$j \longleftarrow nStart + 1$;

while $j < n + 1$ **do**

  while $Pred(j) = \emptyset$ **do**

    if $j \geq (n - nFinish + 1)$ **then**

      $i \longleftarrow random \in \{1, ..., (n - nFinish)\}$;

    end

    else

      $i \longleftarrow random \in \{1, ..., j - 1\}$;

    end

    if $|Succ(i)| < MaxSucc$ **then**

      $Succ(i) \longleftarrow Succ(i) \cup \{j\}$;

      $Pred(j) \longleftarrow \{i\}$;

      $nonredarcs \longleftarrow nonredarcs + 1$;

    end

  end

  $j \longleftarrow j + 1$;

end

Algorithm 2: Generating a precedence network — Step 2.

Data: $nStart, nFinish, MaxPred, nonredarcs$, $Pred(j) : j \in V$, $Succ(j) : j \in V$

Result: Randomly selects a successor u for each non-dummy activity $j \in V : Succ(j) = \emptyset$

$j \longleftarrow n - nFinish$;

while $j > 0$ **do**

  Compute $\overline{Pred}(j)$;

  while $Succ(j) = \emptyset$ **do**

    if $j \leq nStart$ **then**

      $u \longleftarrow random \in \{nStart + 1, ..., n\}$;

    end

    else

      $u \longleftarrow random \in \{j + 1, ..., n\}$;

    end

    if $|Pred(u)| < MaxPred \wedge u \notin \cup_{i \in \overline{Pred}(j)} Succ(i)$ **then**

      $Pred(u) \longleftarrow Pred(u) \cup \{j\}$;

      $Succ(j) \longleftarrow \{u\}$;

      $nonredarcs \longleftarrow nonredarcs + 1$;

    end

  end

  $j \longleftarrow j - 1$;

end

Algorithm 3: Generating a precedence network — Step 3.

Data: $nStart, nFinish, Pred(j) : j \in V, Succ(j) : j \in V, MaxPred, MaxSucc, nonredarcs, NetworkComplexity$

Result: Creates a graph with the desired $NetworkComplexity$

$reqnumarcs := \lceil NetworkComplexity \times (n+1) \rceil$ // required number of arcs to fulfill the desired $NetworkComplexity$;

if $nonredarcs > reqnumarcs$ **then**

  Go to Algorithm 4;

end

else

  while $nonredarcs \leq reqnumarcs$ **do**

    $i \longleftarrow random \in \{1, \ldots, (n - nFinish)\}$;

    if $|Succ(i)| < MaxSucc$ **then**

      if $i \leq nStart$ **then**

        $j \longleftarrow random \in \{nStart + 1, \ldots, n\} \setminus Succ(i)$;

      end

      else

        $j \longleftarrow random \in \{i + 1, \ldots, n\} \setminus Succ(i)$;

      end

    end

    if $|Pred(j)| < MaxPred$ **then**

      if $j \in \overline{Succ(i)} \vee \exists\, u \in \overline{Succ(j)} : Pred(u) \cap \overline{Pred(i)} \neq \emptyset \vee Pred(j) \cap \overline{Pred(i)} \neq \emptyset \vee Succ(i) \cap \overline{Succ(j)}$ **then**

        This arc is not added as it creates redundancy in the network;

      end

      else

        $Pred(j) \longleftarrow Pred(j) \cup \{i\}$;

        $Succ(i) \longleftarrow Succ(i) \cup \{j\}$;

        $nonredarcs \longleftarrow nonredarcs + 1$;

      end

    end

  end

end

Algorithm 4: Generating a precedence network — Step 3 - Removing arcs from the network.

Data: $nFinish, Pred(j) : j \in V, Succ(j) : j \in V, nonredarcs, reqnumarcs$

Result: Removes arcs from the graph until the desired $NetworkComplexity$ is reached

while $nonredarcs > reqnumarcs$ **do**
  $i \longleftarrow random \in \{1, ..., (n - nFinish)\}$;
  if $|Succ(i)| > 1$ **then**
    $j \longleftarrow random \in Succ(i)$;
    if $|Pred(j)| > 1$ **then**
      $Succ(i) \longleftarrow Succ(i) \setminus \{j\}$;
      $Pred(j) \longleftarrow Pred(j) \setminus \{i\}$;
      $nonredarcs \longleftarrow nonredarcs - 1$;
    end
  end
end

Algorithm 5: Activities Generation — Step 2.

Data: $V, \mathcal{L}, \mathcal{L}_j, j \in V, SF$

Result: Skill requirements generation

$\lambda :=$ number of skills required by each activity $j \in V \setminus \{0, n+1\}$;

$\rho :=$ overall number of already associated resources;

if $SF \in]0, 1]$ **then**

  $\lambda \longleftarrow \lceil SF \times L \rceil$;

end

$\mathcal{L}_0, \mathcal{L}_{n+1} \longleftarrow \emptyset$;

while *There is at least one skill not required by any activity* **do**

  $j \longleftarrow 1$;

  $\rho \longleftarrow 0$;

  while $j < n + 1$ **do**

    $\mathcal{L}_j \longleftarrow \emptyset$;

    if $SF \notin]0, 1]$ **then**

      $\lambda \longleftarrow random \in \{2, \ldots, L\}$; //A non-valid SF value has been provided, a variable SF is considered;

    end

    while $|\mathcal{L}_j| < \lambda$ **do**

      $l \longleftarrow random \in \mathcal{L}$;

      if $l \notin \mathcal{L}_j$ **then**

        $\mathcal{L}_j \longleftarrow \mathcal{L}_j \cup \{l\}$;

        $r_{jl} \longleftarrow 1$;

        $\rho \longleftarrow \rho + 1$;

      end

    end

    $j \longleftarrow j + 1$;

  end

end

Algorithm 6: Activities Generation — Step 3.

Data: $V, \mathcal{R}, \mathcal{L}_j, j \in V, MRS, \rho, maxResSkill, maxResAct$

Result: Increase the requirements of the activities until MRS is reached

while $\rho < \left\lfloor \frac{K}{MRS} \right\rfloor$ **do**

  $j \longleftarrow random \in \{1, ..., n\}$;

  if $\sum_{l' \in \mathcal{L}_j} r_{jl'} < maxResAct$ **then**

    $l \longleftarrow random \in \mathcal{L}_j$;

    if $r_{jl} < maxResSkill$ **then**

      $r_{jl} \longleftarrow r_{jl} + 1$;

      if *Activity Skill Requirements are met by solving the associated minimum cost network flow problem* **then**

        $\rho \longleftarrow \rho + 1$;

      end

      else

        $r_{jl} \longleftarrow r_{jl} - 1$;

      end

    end

  end

end