MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Artificial bee colony for project scheduling

Bachelor's Thesis

JAN ŠEVEČEK

Brno, Spring 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Artificial bee colony for project scheduling

Bachelor's Thesis

JAN ŠEVEČEK

Advisor: doc. Mgr. Hana Rudová, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

I hereby declare that this thesis is my own work. All literature and sources of information I used are listed in the list of used literature, and they are properly cited.

Jan Ševeček

Advisor: doc. Mgr. Hana Rudová, Ph.D.

Acknowledgements

I wish to express my deepest gratitude to doc. Mgr. Hana Rudová, Ph.D., for inspiring my interest in the collective intelligence of honey bees, her invaluable insights, guidance, patience and most of all, her time and encouragement. I am grateful to my girlfriend, Nela Pelánková and my family for their full support, especially my mom Michaela Ševečková for sharing her knowledge of Excel for data visualisation, and my brother Pavel Ševeček for his comments about the academic work itself. Additionally, I would like to express my appreciation to Radek Musil for his proofreading and valuable suggestions, which have improved the clarity and accuracy of the English language in this thesis. Their contributions have enriched the quality and impact of this thesis, and I am deeply grateful for their assistance.

Lastly, I would like to extend my thanks to ChatGPT, an AI language model, for providing assistance with data processing. Furthermore, I would like to acknowledge the support of Grammarly, a writing assistance tool, for enhancing the clarity and correctness of my writing.

Abstract

Resource-constrained scheduling problem has many different variants and even more algorithms used in solutions. Multi-skilled resource-constrained project scheduling is a variant adding skills to resources which activities might require. This study deals with this variant with an artificial bee colony, a metaheuristic algorithm inspired by the collective intelligence of honey beehives. The multi-skilled resource-constrained project scheduling, the artificial bee colony algorithms and their implementation are described in detail. Experiments are run on benchmark projects. This thesis shows that the artificial bee colony could be a competitive algorithm in the context of project scheduling.

Keywords

metaheuristics, artificial bee colony, multi-skilled resources, resource-constrained project scheduling

Contents

1 Introduction

Artificial bee colony (ABC) algorithm has gained significant attention as a powerful optimisation technique for solving complex problems [1, 2]. Efficient resource allocation and task assignment are vital for achieving project objectives within constraints in project scheduling [3]. This bachelor's thesis explores the application of the ABC algorithm to project scheduling. It aims to enhance the process by effectively allocating resources, optimising task sequencing, and minimising project completion time. This research investigates the capabilities and performance of the ABC algorithm to contribute to advancing scheduling methodologies and offers insights into practical implementation. The research focuses on the application of the ABC algorithm in project scheduling, specifically the multi-skilled resource-constrained project scheduling problem (MSRCPSP) [4]. This approach provides a unique perspective on project scheduling and offers an alternative to traditional scheduling techniques. By exploring the application of the ABC algorithm in the MSRCPSP, this research aims to evaluate the algorithm's effectiveness in scheduling optimisation problems.

Honey bees, as highly social insects, exhibit complex foraging behaviours, enabling them to gather nectar and pollen from diverse environmental sources efficiently. Of particular interest is their remarkable communication method known as the waggle dance, which allows bees to convey precise information regarding the location of food sources to their fellow hive members. By exploring these fascinating aspects of bee foraging behaviour and communication, we gain valuable insights into the foundational principles that underpin the ABC as a robust optimisation technique.

The thesis is structured as follows. Chapter 2 first discusses the resource constrained project scheduling problem (RCPSP) in general. Then, the MSRCPSP is introduced and explored in detail, including a mathematical model.

Chapter 3 describes various methods for solving RCPSP. It emphasizes algorithms based on the behaviour of honey bees. More specifically, the algorithms based on the foraging behaviour and communication of the honey bees are described. Then the ABC is introduced,

including a general pseudocode for the algorithm. The application of the ABC on the project scheduling problems is also discussed.

Chapter 4 examines the specific implementation of the ABC algorithm for the MSRCPSP created for this thesis. This chapter consists of two parts. One focuses on the data structures of the implementation, while the other on the algorithms used for solving the problem.

Finally, Chapter 5 discusses the experiments run with the ABC applied on the MSRCPSP benchmarks. The benchmarks' specifications and the selection process of the instances for experiments are described. Finally, parameter tuning and the results of the experiment runs are discussed.

2 Project scheduling problem

This chapter describes the basic problem and its variation. First, a general description is provided. Then, a more specific characterisation of the basic RCPSP is provided, and the multiskilled variation of the RCPSP is introduced and discussed in more detail. Lastly, a few solution approaches are provided for the problem.

2.1 Problem description

RCPSP is a well-known and important problem in project management and operations research [5]. The goal of an RCPSP is to schedule a set of activities that need to be completed in a project, subject to limited resources and precedence constraints. The objective can be, for example, to find a schedule that minimises the *makespan*, which is the project's total duration from start to completion. But other objectives could also be to minimise cost, some other criteria or a weighted combination of these. The RCPSP has many variations and modifications to many aspects, such as activity preemption and multi-mode activities or resource variation like nonrenewable and partially renewable resources, cumulative resources or resources with multiple skills.

A multi-skilled resource-constrained project scheduling problem (MSRCPSP) is an extension of the RCPSP where resources are not only limited in quantity but also in skill set [6]. Each activity requires several resources possessing various skills for it to be completed, and each resource possesses a set of skills it can provide. In addition to resource availability, the scheduling must also consider the skill compatibility between activities and resources, meaning which resources master the required skills to use for an activity to satisfy its skill needs. Finding an optimal schedule that satisfies both resource availability and skill compatibility is challenging and has received much attention in recent years [7].

2.1.1 Resource-constrained project scheduling

RCPSP contains a set of activities $A = \{0, 1, 2, ..., n, n+1\}$, where activities 0 and $n+1$ are so-called *dummy* activities representing

the start of a project and the end respectively, and set of resources $R = \{1, 2, ..., m\}$. Each activity $j \in A$ has a processing time p_j, precedence conditions $Prec_j$, and for each resource i, it has resource requirement $r_{j,i}$. Processing time p_j describes how long it takes to finish activity j once it starts. Precedence conditions $Prec_j$ is a set of activities that must be finished before activity j can be started. And finally, resource requirement $r_{j,i}$ expresses how many resources i activity j needs to be processed. Dummy activities 0 and $n + 1$ have processing time equal to zero, denoted as $p_0 = p_{n+1} = 0$, and they do not require any resources so $r_{0,i} = r_{n+1,i} = 0$ for all $i \in R$. In basic RCPSP, we work without preemption, the activity has to finish once it started. Each resource $i \in R$ has a capacity cap_i, which is how many resources i are available at each moment. In addition, resource usage for each resource $i \in R$ cannot exceed its capacity cap_i at any given moment.

Decision variables for starting time S_j are defined for each activity $j \in A$. This also gives us completion time $C_j = S_j + p_j$. The goal is to find a schedule where the objective function is optimised and all constraints are satisfied. The scheduled activities are subject to precedence constraints and resource constraints. The objective function is the minimalisation of makespan $C_{\max}$. Makespan is the maximum C_j for all $j \in A$. Alternatively, the objective function can be other criteria, as stated at the beginning of this section.

2.1.2 Multi-skilled resource-constrained project scheduling

In a multi-skilled resource-constrained project scheduling problem (MSRCPSP), we have the same set of activities $A = \{0, 1, 2, ..., n, n + 1\}$, resources $R = \{1, 2, ..., m\}$, processing times p_j, precedence constraints $Prec_j$ for each activity $j \in A$ as in a basic RCPSP [6]. However, in the MSRCPSP, each resource is associated with a set of skills, and each activity requires a specific number of resources having the required skill for each skill to be performed. Each resource can only use one skill for at most one activity at any given time.

Let K be the set of skills and let $r_{j,k}$ denote how many resources possessing skill $k \in K$ are required by activity j. In the MSRCPSP, each resource $i \in R$ is associated with a set of skills that it possesses, denoted as K_i.

Table 2.1: Model notation specifications

Parameters	p_j	Processing time of activity j
	$r_{j,k}$	Number of resources having skill k required by activity j
Sets	A	Set of activities
	R	Set of resources
	K	Set of skills
	K_i	Set of skills mastered by resource i
	$Prec_j$	Set activities, that precedes activity j
Decision variables	S_j	starting time of activity j
	C_j	Completion time of activity j
	$X_{i,k,j}$	1, if resource i uses skill k for activity j, 0 otherwise
	$Y_{j,j'}$	1, if activity j is completed, before activity j' is started, 0 otherwise

A few decision variables are also defined. First is the starting time S_j for each activity j. From the starting time S_j and processing time p_j, completion time is defined as $C_j = S_j + p_j$. Decision variable $X_{i,k,j}$ equals 1 if resource i uses skill k to perform activity j. Otherwise, it is equal to 0. Lastly, the decision variable $Y_{j,j'}$ equals 1 if activity j is completed before activity j' is started and is equal to 0 otherwise. **Table 2.1** summarises the notation used by the mathematical model of this problem.

The goal is to optimise the objective function, which is, in our case, the minimisation of the makespan $C_{\max}$ of the project. The mathematical model, inspired by that of Jakob Snauwaert and Mario Vanhoucke [6], is summarised by the following:

$$\min \quad C_{\max} \tag{2.1}$$

$$s.t. \quad C_{j'} \leq S_j \qquad \forall j \in A, j' \in Prec_j \tag{2.2}$$

$$\sum_{i \in R} X_{i,k,j} = r_{j,k} \qquad \forall j \in A, k \in K \tag{2.3}$$

$$\sum_{k \in K} X_{i,k,j} \leq 1 \qquad \forall j \in A, i \in R \tag{2.4}$$

$$Y_{j,j'} + Y_{j',j} \leq 1 \qquad \forall j, j' \in A : j < j' \tag{2.5}$$

$$\sum_{k \in K} X_{i,k,j} + X_{i,k,j'} \leq 1 + Y_{j,j'} + Y_{j',j} \qquad \forall j, j' \in A : j < j', i \in R \tag{2.6}$$

$$X_{i,k,j} \leq \begin{cases} 1, & \text{if } k \in K_i \\ 0, & \text{otherwise} \end{cases} \qquad \forall j \in A, i \in R, k \in K \tag{2.7}$$

$$C_j = S_j + p_j \qquad \forall j \in A, S_j, C_j \in \mathbb{N} \tag{2.8}$$

$$S_j \geq 0 \qquad \forall j \in A \tag{2.9}$$

$$X_{i,k,j} \in \{0,1\} \qquad \forall j \in A, k \in K, i \in R \tag{2.10}$$

$$Y_{j,j'} \in \{0,1\} \qquad \forall j, j' \in A : j \neq j' \tag{2.11}$$

The goal is denoted as minimising the makespan, $C_{\max}$ (2.1), the completion time of the last activity. The precedence constraints are described by (2.2). Resource-skill constraints required by each activity are specified by (2.3). Every resource can provide at most one skill per activity as specified by (2.4). The constraints (2.5) outline the connection between all decision variables $Y_{j,j'}$. $Y_{j,j'} + Y_{j',j}$ either equals one if the activities do not overlap (one ends before the other one begins) or zero if they do overlap (they run in parallel for some time). The constraints (2.6) ensure that two activities scheduled in parallel do not use the same resource. The constraints (2.7) specify that the resource can only perform a skill it masters. The constraints (2.9) define the domain of decision variable S_j, (2.10) the domain of $X_{i,k,j}$ and finally (2.11) the domain of $Y_{j,j'}$.

3 Solution methods

This chapter discusses the solution methods for multi-skilled resource-constrained project scheduling problem (MSRCPSP) while focusing on various aspects of the behaviour of honeybees in nature and algorithms inspired by them. First, a general overview is provided, and then the focus will be on honeybees, especially the foraging and communication of honeybees and computational methods based on this phenomenon. Lastly, one of the methods, the artificial bee colony (ABC), is described in more detail. This algorithm is used as a basis for this thesis.

Multiple methods have been applied to solve this problem [7]. Exact methods like linear programming, integer linear programming, mixed integer programming or constrained programming have successfully solved project scheduling problems. Due to the limitations of the exact methods in terms of time and space requirements for the solution, many approaches use *metaheuristics* to solve this NP-hard problem.

Metaheuristics are of two base categories [8]. First, there are the single solution metaheuristics. The main idea behind this type is the improvement of a single solution. These algorithms search locally in the neighbourhood of the solution, performing a few operators to modify it and selecting a solution from candidates for the next iteration. Often, single-solution metaheuristics do not get a better solution in every single iteration from the last one, like simulated annealing [8]. For this thesis, population-based metaheuristics are more relevant.

Population-based algorithms operate on a set of solutions, unlike metaheuristics which work on a single solution. One category of the more known population-based metaheuristics is evolutionary algorithms. The well-known genetic algorithms are in this category [9]. In the genetic algorithm, each encoded solution is called a chromosome. The algorithm creates an initial population (generation). Then it selects some parents and applies operators like *mutation* or *crossover* to generate offspring and select the next generation's population. The next generation is usually chosen based on *fitness*. Fitness is the value of the objective function. More fitness means a better solution.

Population-based metaheuristics are also represented by algorithms based on the social behaviour of various creatures. The most famous of these is ant colony optimisation [10], inspired by the behaviour of the ants and their ability to communicate using pheromones. These metaheuristics work with a population of solutions. Each solution is tried to be improved by following simple independent rules while sharing knowledge with others. Other examples are the flocking of birds, movements of schools of fish [11], or the algorithms based on honeybees studied in this thesis [2, 12, 13, 14, 15].

3.1 Bee algorithms

Multiple algorithms have been inspired by the curious behaviour of honeybees [12]. Many aspects of their behaviour have been studied and used as a basis for various metaheuristics. Among them are methods based on queen bee mating and marriage, such as queen bee evolution, marriage in honey bee optimisation or honey bees mating optimisation algorithm. These algorithms primarily focus on the queen bee and its evolution, working in terms of chromosomes similarly to genetic algorithm [9]. Others are built on swarming, navigation and spatial memory, or division of labour. For this thesis, however, the most important are the methods based on communication and foraging.

The communication and foraging behaviour of honeybees inspired several optimisation algorithms. Honeybees search space around the hive to find nectar or other material the hive needs. Once they find such a commodity, they return to the hive's dance floor and perform a waggle dance. This waggle dance is used to communicate with other bees, sharing the resource's location and quality. Most, if not all, of the algorithms in this category are based on this phenomenon.

One of the first algorithms in this family was the bee system (BS) [13]. Sato and Hagiwara first proposed it in 1997. In this approach, each bee represents a chromosome (solution). The bees associated with chromosomes with high fitness are considered superior, and other bees generate multiple solutions by searching around the vicinity of the superior ones. The BS and its variants have been mainly applied to the travelling salesman problem. Other similar, more gen-

eralised, and improved algorithms, such as bee colony optimisation, exist [12]. In this algorithm, there are two processes: forward pass and backward pass. In the forward pass, the bees try to improve their solutions by modifying them several times, while in the backward pass, they share the information with their hive mates. Then they might try to recruit them to search for a new solution around theirs or abandon their solutions and initialise a new one randomly or in the vicinity of another bee. Another example would be the bees algorithm [12], where the population of bees is divided into two groups: scouts and recruits. While scouts are tasked with exploring the search space, the recruits exploit found solutions.

3.1.1 Artificial bee colony optimisation

The most relevant algorithm for this thesis is the artificial bee colony (ABC), which was introduced by Karaboga and Basturk [1, 2]. In the ABC algorithm, a colony of artificial bees searches for the optimal solution in the search space by mimicking the foraging behaviour of honeybees. The algorithm has been widely used in various optimisation problems and has shown good performance in terms of solution quality and convergence speed. In ABC, solutions are called *food sources*, and there are three types of bees. Bees associated with one food source trying to exploit this food source are called *employed bees*. Bees exploiting a selected food source from employed bees based on their fitness are called *onlooker bees*. And bees, which fail to improve their food source given several iterations, become *scout bees* and search for new solutions randomly, thus exploring the search space. When a scout bee finds a better food source, it ceases to be a scout and returns to its original role as an employed or onlooker bee.

The algorithm (see **Algorithm 1**) first has to set up all the parameters, like the number of employed bees $NoEmployed$ and the number of onlooker bees $NoOnlookers$, the limit $Limit$ of how many times a bee can fail to improve its solution until it becomes scout. Then food sources are initialised randomly (line 2). Trials for all bees are set to zero (line 3). The algorithm iterates (lines 4-9) until a satisfaction criterion is met, which could be, for example, a certain number of iterations or a certain number of consecutive iterations that did not produce an improving solution. An iteration consists of three phases,

one for each type of bee (lines 5-7). At the end of each iteration (line 8), the best food source found so far is saved. When the iterations are over, the best food source *BestFoodSource* is returned (line 10).

Algorithm 1 Artificial bee colony

```
1:  function ABC(NoEmployed, NoOnlookers, Limit)
2:      InitialiseFoodSourcesRandomly
3:      InitialiseTrialsForAllBeesToZero
4:      repeat
5:          SendEmployedBees(EmployedBees)
6:          SendOnlookerBees(OnlookerBees, EmployedBees)
7:          SendScoutBees(ScoutBees)
8:          BestFoodSource ← GetBestSolutionSoFar
9:      until satisfaction criteria is met
10:     return BestFoodSource
```

First, the employed bees (see **Algorithm 2**) are sent to apply an operator to modify their solution. By doing so, they search locally for a better solution next to the one they are associated with. They create a modified solution and evaluate its fitness (line 3). If the solution is better than the current one(line 4), they remember the new solution instead (line 5). Otherwise, increase the number of trials $Trial_{eb}$ on this food source (line 7). If the number of trials exceeds the limit (line 8), it notes its previous role as an employed bee (line 9) and becomes a scout (line 10).

Second, the onlooker bees (see **Algorithm 3**) each select a food source from the employed bees (line 3). The higher the fitness, the more probable it will be chosen. This can be realised by *roulette wheel*, *tournament selection*, or by other such techniques [16]. Again, similarly to employed bees, they create a modified solution and evaluate its fitness (line 4). If the solution is better than the current one (line 5), they remember the new solution instead (line 6). Otherwise, increase the number of trials $Trial_{ob}$ on this food source (line 8). If the number of trials exceeds the limit (line 9), it notes its previous role as an onlooker bee (line 10) and becomes a scout (line 11).

Third, the scout bees (see **Algorithm 4**) initialise a new food source by generating a new solution randomly (line 3). Remember the new

Algorithm 2 Employed bees

```
1:  procedure SendEmployedBees(EmployedBees)
2:      for each eb ∈ EmployedBees do
3:          FoodSource* ← ModifyAndEvalFS(FoodSource_eb)
4:          if FoodSource* is better than FoodSource_eb then
5:              FoodSource_eb ← FoodSource*
6:          else
7:              Trial_eb ← Trial_eb + 1
8:              if Trial_eb > Limit then
9:                  PrevRole_eb ← Employed
10:                 BecomeScout(eb)
```

Algorithm 3 Onlooker bees

```
1:  procedure SendOnlookerBees(OnlookerBees, EmployedBees)
2:      for each ob ∈ OnlookerBees do
3:          FoodSource* ← SelectFSBasedOnFitness(EmployedBees)
4:          FoodSource** ←ModifyAndEvalFS(FoodSource*)
5:          if FoodSource** is better than FoodSource_ob then
6:              FoodSource_ob ← FoodSource**
7:          else
8:              Trial_ob ← Trial_ob + 1
9:              if Trial_ob > Limit then
10:                 PrevRole_ob ← Onlooker
11:                 BecomeScout(ob)
```

solution one (line 4), return to its original role (line 5) and set its trial ($Trial_{sb}$) to 0 (line 6).

3.2 ABC for project scheduling

This section summarises related work concerning the ABC and MSRCPSP. First, an application of ABC on basic RCPSP is described. Then a discrete variation of the ABC is discussed. And finally, other algorithm solving MSRCPSP is shown with a focus on relevant aspects of the problem, which could be used in this thesis.

Algorithm 4 Scout bees

```
1: procedure SendScoutBees(ScoutBees)
2:     for each sb ∈ ScoutBees do
3:         FoodSource* ← InitialiseNewFSRandomly
4:         FoodSource_sb ← FoodSource*
5:         ReturnToPreviousRole(sb, PrevRole_sb)
6:         Trial_sb ← 0
```

3.2.1 ABC in solving of RCPSP

In 2011 Reza Akbari et al. [14] proposed a continuous solution to RCPSP using ABC. This paper presents a model for a solution encoded as a vector with *priority values*. The higher the priority value is, the sooner the activity will be scheduled if precedence constraints are met. A project with n activities is encoded in a vector with n priority values between 0 and 1. This paper uses a serial schedule generation scheme (serial-SGS) to decode the schedule. The activities that meet the precedence conditions are scheduled one by one in accordance with their value in the vector. To modify the food source, the bee selects a random bee as its neighbour and modifies one dimension in its position based on the neighbour.

In 2013, a discrete solution was proposed by Nouha Nouri et al. [15]. This paper encodes the solution as a *priority list* instead of a vector with priority values. Unlike a vector with priority values, the position in the list is the rule for prioritising activities for scheduling. The lower the index is, the sooner the activity should be scheduled in the decoding scheme when the precedence constraints are met. In this paper, a serial-SGS is also used to create a schedule. For the basis of modification of the solution, one or two *insert* or *swap* operators are performed. In addition, the so-called adaptive mechanism is used to select which operator to choose to improve the quality of the solution. The results surpassed the continuous ABC solution.

3.2.2 MSRCPSP

One of the relevant papers about the MSRCPSP is from Jian Lin et al. [17] published in 2020. Their approach is to use genetic program-

ming to solve this problem. The existence of the skills extends the encoding of the solution and has to account for the issue of choosing from multiple resources for a skill, which they master, for an activity. This is achieved by expanding the priority list encoding used in a discrete variation of the ABC for the RCPSP with a priority list of resources for each activity. When decoding the solution, the resources with lower indices in the list are tried to be used for their skills first. This is the relevant aspect of this paper for this thesis. The decoding is a *repair-based generation* of feasible solutions. Decoding the solution can produce an invalid schedule, violating a constraint, which is then modified to a valid one. The modifications of the solutions are made by *crossovers* and *mutations*.

4 Solution approach

This chapter discusses the solution approach of this thesis for the multi-skilled resource-constrained project scheduling problem (MSRCPSP) using artificial bee colony (ABC). The implementation is written in C#. Data structures are introduced and discussed in the first section. In the second section, the algorithms of the implementation are described.

4.1 Data structures

In this section, first, the parsed project structure is briefly discussed. Then the main data structures representing encoded solutions known as *food sources* are described. Lastly, decoded solutions known as *schedule S* are presented next.

4.1.1 Project

The project contains basic information about the project, denoted as P, namely the number of activities, resources and skills. Apart from that, a project contains a list of parsed activities and a list of parsed resources. Activities and resources are represented as objects. Skill types are represented as numbers.

Each activity contains information about its identity, processing time, predecessors (activities that must be completed to process this activity) and an array of skill requirements. In this array, the index symbolises the identification of the skill and the value of the number of resources mastering said skill this activity needs to be processed.

Each resource has a unique identifier. In addition, the resource has an array of skills it masters. Similarly to the activity, the index represents a skill type, while the value represents if the resource masters a skill or not.

4.1.2 Engine

The main structure for the run of this ABC implementation is the structure *Engine*. The engine contains information about the project and two lists of food sources – one for employed and onlooker bees,

denoted as *FoodSources*, and one for scout bees, denoted as *ScoutBees*. In our case, the bees are virtual, meaning they do not have any class of their own and are only represented by the linked lists of food sources and the way the lists are used. In addition, the engine saves the input parameters: the number of food sources $NoFS$, the maximum number of iterations, the maximum number of consequent iterations without an improvement and the limit *Limit* of unsuccessful trials to generate a new food source.

4.1.3 Food source

The food source, denoted as A, is the encoded solution for the ABC. It contains the priority linked list of encoded activities and the number of trials it has been unsuccessfully tried to be improved. The encoded activity is an activity as in the project structure, but it is extended with a resource priority linked list. This new list of resources subsequently determines the order in which the available resources are attempted to be utilised for processing this activity. Similarly to the resources' priority, the linked list of the activities is used in scheduling. The activities are scheduled from the head of the list. The activities in this list are always kept semi-ordered in terms of precedence. This means that for all activities in the list, their predecessors come before them (having a lower index).

4.1.4 Schedule

The schedule is the decoded solution. It contains a list of scheduled activities and a two-dimensional structure representing resource usage of activities in time. Scheduled activities are activities with an assigned starting time, calculated completion time and a dictionary of resources to skill assignment used to process the activity. The dictionary uses resources as keys to represent the chosen resources for activity processing, while the corresponding skills required for the activity are represented by numbers.

The resource usage structure has time as one dimension and resources as the second. Its values are identifiers of activities that use a given resource at a given time. This structure is used mainly for scheduling activities, specifically to check if a resource is used at a

given time or is available for other activities. But it also represents the schedule for printing to a console or file.

4.2 Algorithms

This section introduces all algorithms of the implemented ABC. First, the food source initialisation is discussed. Then the operators for modifications of the food sources are described. Next, creating a schedule (evaluating the food sources; creating a decoded solution from an encoded one) is discussed. Last, the engine, core of the ABC, the algorithm itself is described as a whole.

In the beginning, a few helper functions are defined. One of them is function Predecessors(P, a) returns the predecessors of the activity a in a project P. Successors(P, a) returns the successors of the activity a in a project P. Also, on the priority list A, two functions Next(A, a) and Previous(A, a) are defined. Next(A, a) returns the next activity (activity with the index of activity a plus 1) in the list. Previous(A, a) returns the previous activity (activity with the index of activity a minus 1) respectively.

4.2.1 FoodSource initialisation

The initialisation of a food source is the process of creating a new one at the beginning of the ABC algorithm. New food sources are also created in this way in the scout bee phase in the ABC algorithm after the old food sources are discarded.

The initialisation of the food sources (see **Algorithm 5**) takes a list of project activities A and project with information P of all these activities. It first shuffles the list of activities except for the first dummy activity a_0 in random order into a list $Shuffled Activities$ (line 2). A new list $Reordered Activities$ with only the first activity a_0 is initiated (line 3). This new reordered list is progressively constructed (lines 4-9). As a result, the list only consists of activities with all their predecessors in this list before them. The construction process selects an activity a in a cycle (lines 5-9). An activity is selected if $Reordered Activities$ already contains all its predecessors (line 6). The activity is then removed from $Shuffled Activities$ and added to $Reordered Activities$ (lines 7-8). The

selection process is then run again, and new activity is selected until the $ShuffledActivities$ list is empty and $ReorderedActivities$ contains all activities (lines 4-9). As a result, a new reordered list of activities is returned (line 10). This list is the priority list [1] of a new food source. Each activity has its resource priority list assigned at random.

Algorithm 5 Food source initialisation

```
1:  function InitFoodSource(P)
2:      ShuffledActivities ← Shuffle(Activities(P) \ {a_0})
3:      ReorderedActivities ← [a_0]
4:      while |ShuffledActivities| > 0 do
5:          for each a ∈ ShuffledActivities do
6:              if Predecessors(P, a) ⊆ ReorderedActivities then
7:                  ShuffledActivities ← ShuffledActivities \ {a}
8:                  ReorderedActivities ← ReorderedActivities ∪ {a}
9:                  break
10:     return ReorderedActivities
```

4.2.2 Operators

In the ABC algorithm during the employed and onlooker phases in the ModifyAndEvalFS (see **Algorithm 2** on page 11 and **Algorithm 3** on page 11) function an operator is selected randomly and used to modify the food source. Operators are used for modifying food sources. This modification is used during the employed bee and the onlooker bee phase. If the modified food source is better than the unmodified one (has a lower makespan after evaluation), the new food source replaces the old one. Two operators are implemented: insert activity and insert resource.

Insert activity

The idea behind this operator is simple. It modifies the food source by picking an activity in the activities' priority list and inserting it

1. Note that set operations are used for simplicity, even though it is an ordered structure. The same notation is kept for other structures where the ordering is clear.

into another place in the list while preserving the precedence constraints (for all activities in the priority list, all their predecessors are in the list before them). First, an activity a has to be selected. New potential places for insertion have to be found. This can be achieved by searching the priority list from the chosen activity to the first one until a predecessor is found. These activities, represented in a set $Before$, represent activities in the list before which selected activity can be inserted. Similarly, the list can be searched from the chosen activity to the last one until a successor is found. These activities represent the activities, denoted as set $After$, after which the selected activity can be inserted. These two created lists are candidates for insertion, so the integrity of the precedence constraint of the list is preserved. And finally, a new place for insertion has to be selected from candidates, and the activity inserted into the new place in the list.

The activity insertion (see **Algorithm 6**) picks a non-dummy activity a in the food source's priority list (line 2). A new place for the activity is found as follows. Two sets are constructed.

The set $Before$ is constructed from the activities in the priority list before a until a predecessor of activity a is found (lines 3-7). $Before$ represents activities before which activity a can be inserted so the precedence constraints are preserved. Previous activity a' before a in A is selected. While it is not a predecessor of the activity a, it is added to the set $Before$, and the previous activity is chosen as a'. This runs in a loop until a' is a predecessor of a.

Similarly, the second set $After$ is constructed as well. This time activities after a in the priority list A are searched and added to the set $After$ until a successor of the activity a is found (lines 8-12). The set $After$ represents activities after which a can be inserted while precedence in the priority list will be preserved. This time next activity a'' after activity a is selected. If a'' is not a successor of a, it is added to the set $After$. In a loop, previous activities a'' are chosen and added to the set $After$ until a'' is a successor of a.

Now, the potential places for insertion of activity a in the sets $Before$ and $After$ are constructed. The algorithm must check if there is a possible place for insertion (line 13). Both sets $Before$, and $After$ might be empty. If so, the algorithm runs again from the beginning, selecting a new activity a for insertion (line 14). This happens when its predecessor and successor immediately surround selected activity a.

Algorithm 6 Insert activity

```
1:  function InsertActivity(A, P)
2:      a ← PickActivityAtRandom(A \ {a_0, a_{n+1}})
3:      Before ← ∅
4:      a' ← Previous(A, a)
5:      while a' ∉ Predecessors(P, a) do
6:          Before ← Before ∪ {a'}
7:          a' ← Previous(A, a')
8:      After ← ∅
9:      a'' ← Next(A, a)
10:     while a'' ∉ Successors(P, a) do
11:         After ← After ∪ {a''}
12:         a'' ← Next(A, a'')
13:     if |Before| + |After| == 0 then
14:         return InsertActivity(A, P)
15:     Remove(A, a)
16:     if Random(0, 1) < |Before| / (|Before|+|After|) then
17:         ā ← PickActivityAtRandom(Before)
18:         AddBefore(A, a, ā)
19:     else
20:         ā ← PickActivityAtRandom(After)
21:         AddAfter(A, a, ā)
22:     return A
```

Activity a is then removed from the food source A (line 15) and inserted into a different place in the priority list. A selection of the place for insertion is based on the constructed lists at random. First, either $Before$ or $After$ is randomly selected based on their lengths (line 16). The following equation shows the probability of selecting $Before$ (and vice versa for $After$):

$$P(Before) = \frac{|Before|}{|Before| + |After|} \tag{4.1}$$

Once a set is selected, the activity $\bar{a}$ in the list is chosen randomly (lines 17 or 20). The activity a is then inserted into a priority list A

before an activity $\bar{a}$ selected from the set $Before$ (lines 18) or after an activity chosen from the set $After$ respectively (lines 21). A new modified priority list A is returned (line 22).

Insert resource

The idea behind the operator for the resource insertion can be summarised as follows. Modifying the food source by changing the resource priority list of an activity in its activities' priority list requires a few things. First, it requires the selection of an activity from the activities' priority list. Then it needs to select a resource in the resource priority list of the chosen activity. Finally, it has to select the new place for the selected resource in the resource priority list and insert it into its new place.

The insertion of a resource (see **Algorithm 7**) is similar to activity insertion. First, a non-dummy activity a is randomly selected (line 2). The resources' priority list R associated with this activity is selected for modification (line 3). A resource r in this priority list is selected randomly (line 4), and a new place, different from the current one, is selected for insertion (lines 5-13). The two sets $Before$, and $After$ are constructed (lines 5-6). The difference from the activity insertion in the activities' priority list is the precedence constraints. In the resources' priority list, there are no such constraints. This means the construction of sets $Before$ and $After$ is trivial. The resource r is removed from the resource priority list R (line 7). Similarly to activity insertion, the new place is selected from the sets $Before$ and $After$, and the resource r is inserted into its new place in the list (lines 8-13). The new resource priority list is then returned (line 14).

4.2.3 Create schedule

Creating a schedule by decoding encoded solution – food source – has to find starting times and assign appropriate resources with needed skills for each activity. This is achieved by scheduling activities one by one by their order in the activities' priority list. There are two constraints for a scheduled activity. One is the precedence constraint. A scheduled activity must start after all its predecessors have finished. This gives a lower bound for its starting time. The second constraint

Algorithm 7 Insert resource

```
1:  function InsertResource(A)
2:      a ← PickActivityAtRandom(A \ {a_0, a_{n+1}})
3:      R ← GetResourcePriorityList(A, a)
4:      r ← PickResourceAtRandom(R)
5:      Before ← GetResourcesWithLowerIndexInList(R, r)
6:      After ← GetResourcesWithHigherIndexInList(R, r)
7:      Remove(R, r)
8:      if Random(0, 1) < |Before| / (|Before|+|After|) then
9:          r' ← PickActivityAtRandom(Before)
10:         AddBefore(R, r, r')
11:     else
12:         r' ← PickActivityAtRandom(After)
13:         AddAfter(R, r, r')
14:     return A
```

is the resource-skill constraint. Each activity must have all its skill requirements satisfied. So enough resources that master appropriate skills have to be available for its processing. Minimising the makespan of the entire project requires each activity to be scheduled as soon as possible.

Decoding a solution from an encoded one works as follows (see **Algorithm 8**). A new schedule *Scheduled* (activity-starting time assignment) is created with the first dummy activity at time 0 (lines 2-3). Then the algorithm picks each activity a from the food source's priority list in a cycle (lines 4-17). Finding the starting time for each activity a works in two phases.

The first phase finds a minimal starting time S_a to meet precedence constraints without considering resources (lines 5-8). All predecessors are scheduled already at this point. That is ensured by the nature of the activities' priority list. All predecessors of each activity are in the list before it. The maximum completion time from all predecessors is calculated as the lower bound for the starting time S_a of a (line 8).

The second phase accounts for the resource-skill assignment (lines 10-19). The algorithm checks if it is possible to satisfy the skill requirements of the activity a starting at the lower bound starting time S_a.

Algorithm 8 Creating schedule

```
1:  function CreateSchedule(A, P)
2:      Scheduled ← ∅
3:      Scheduled[a_0] ← 0
4:      for each a ∈ A \ {a_0} do
5:          p_a ← GetProcessingTime(a)
6:          for each a' ∈ Predecessors(P, a) do
7:              p_a' ← GetProcessingTime(a')
8:              S_a ← max(S_a, Scheduled[a'] + p_a')
9:          R ← GetResourcePriorityList(a)
10:         while not Assigned(S_a) do
11:             for i ∈ [1 ... m] do
12:                 ResetSkillAssignment(S_a)
13:                 for each r ∈ R do
14:                     if IsResourceAvailable(r, S_a, S_a + p_a) then
15:                         AssignSkillForActivityAtRandom(A, a, r)
16:                         if AllSkillsSatisfied(a) then
17:                             Add(Scheduled, S_a)
18:                             break
19:             S_a + +
20:     return Scheduled
```

The lower bound starting time is incremented if the activity is not successfully scheduled and the activity is tried to be scheduled again. At the beginning of each iteration, the algorithm resets the skill-resource assignment (line 12). This mainly aims to reset a failed assignment if the previous iteration didn't find a starting time for activity a. The algorithm works for each resource r in the resource priority list of a as follows (lines 13-18). If the resource r is available from the lower bound starting time to the completion time and masters a required skill by a, it is selected (lines 14-15). If all skills are satisfied at this point (line 16), the a is added to the schedule at time S_a (line 17), and a new activity for scheduling is selected and scheduled (line 18). Otherwise, once all resources in the resource priority list have been cycled through, starting time estimate is incremented (line 19), and the resources cycle runs again. Since the resource-to-skill assignment

works completely randomly, running the cycle multiple times might be good before incrementing the starting time estimate (lines 11-19). The found schedule is returned once all activities are scheduled (line 20).

4.2.4 Engine

The ABC algorithm itself (see **Algorithm 9**) is mainly implemented as described in Section 3.1.1. However, there are a few differences. The most important one is that the employed bees and onlookers share the same food sources. As a result, the number of food sources $NoFS$ equals the number of employed bees and the number of onlookers. This means effectively that there are two lists of food sources, $FoodSources$ and $ScoutBees$.

Algorithm 9 Artificial bee colony – engine implementation

```
1:  function ABC(P, NoFS, Limit)
2:      FoodSources ← InitMultipleFoodSources(P, NoFS)
3:      InitialiseTrialsForAllBeesToZero(FoodSources)
4:      repeat
5:          SendEmployedBees(FoodSources)
6:          SendOnlookerBees(FoodSources, FoodSources)
7:          SendScoutBees(ScoutBees)
8:          BestFoodSource ← GetBestSolutionSoFar(FoodSources)
9:      until MaxIterationReached() or
    IterationsWithoutImprovementReached()
10:     return BestFoodSource
```

In the employed bee phase, each food source from $FoodSources$ generates a new solution in its neighbourhood (line 5). If the new food source is better than the current one, the new food source is remembered instead. If the food source is not improved for a certain amount of iterations $Limit$, the food source is removed from $FoodSources$ and added to the $ScoutBees$. The implementation logically follows **Algorithm 2** on page 11.

In the onlooker bee phase, the probability of selecting each food source in the roulette wheel is calculated (line 6). For each food source, another food source is selected with the roulette wheel. A new food source is generated in the neighbourhood of the selected food source.

If the generated food source is better, it replaces the current one. If the selected food source has not been improved for the given number of iterations *Limit*, the food source becomes scout as in the employed bee phase. For pseudocode, see **Algorithm 3** on page 11.

In the scout bee phase, for each food source from the *ScoutBees*, a new food source is initialised instead (line 7). It is removed from *ScoutBees* and added to *FoodSources*, similarly to **Algorithm 4** on page 12.

Each iteration saves the best food source *BestFoodSource* (line 8). The cycle is terminated if either of the stopping criteria is reached (line 9). Finally, the *BestFoodSource* is returned (line 10).

5 Experiments

This chapter presents the experiments conducted to evaluate the algorithm's performance on the multi-skilled resource-constrained project scheduling problem (MSRCPSP). The implementation is in C#. First, a brief summary of the machine the experiments ran is introduced. A description of the benchmarks and project metrics used for their generation is provided, along with the instance sets tested and the algorithm parameters tuned during the experiments. Finally, the results are discussed, and a comparison with the quality of solutions from previous studies is presented. The experiments in this bachelor thesis were conducted on a machine with the following specifications:

1. Hardware Specifications
    - Processor: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz, 3201 Mhz, 4 Cores, 4 Logical Processors
    - Memory: 24 GB DDR3 RAM, 2400 MHz.
2. Software Specifications
    - Operating System: Microsoft Windows 10 Pro
    - Programming Language(s): C#
    - Libraries and Frameworks: .NET

5.1 Benchmarks

This section offers a general description of the benchmarks utilised in the experiments. The benchmarks are from Snauwaert and Vanhoucke (2023) [6], published on website [18]. These benchmarks contain instances with varying numbers of activities, such as 30, 60, and 90 activities, encompassing a wide range of project sizes and complexities.

5.1.1 Project metrics

The following project metrics were used as input for the benchmark generation [6].

- *Skill factor*: The skill factor metric measures the relationship between the skills required for an activity or project and the skills the resources possess. It considers both the activity level (SF_j) and the project level (SF). A higher skill factor indicates a higher level of resource competence in relation to the project's skill requirements, thus defining the density of the skill requirements matrix. This concept is proposed in Correia et al. (2012) [19] and offers insights into the alignment between project demands and resource capabilities.

  Skill factor for an activity j (SF_j):

$$SF_j = \frac{\sum_{k \in K} b_{j,k}}{|K|} \tag{5.1}$$

  Binary variable $b_{j,k}$ equals 1 if activity j requires skill k for its completion. In other words, it equals 1 if $r_{j,k}$ is greater or equal to 1 and is 0 otherwise.

$$b_{j,k} = \begin{cases} 1, & \text{if } r_{j,k} \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

  Skill factor for the project (SF):

$$SF = \frac{\sum_{j \in A} SF_j}{|A|} \tag{5.3}$$

- *Skill strength*: The skill strength metric evaluates the overall expertise of resources available concerning specific skills and the entire project. It is calculated by comparing the occurrence of a skill in the workforce to the amount of the skill required in the project. A higher skill strength implies a higher level of resource competence. The skill strength represents the scarcity of skill in a project. The skill strength can be defined for both individual skills (SS_k) and the project level (SS). The project skill strength measures the total skill scarceness in the project, while the skill strength for individual skills can vary across different skills. This

metric is essential for analysing the alignment between project demands and the expertise of the available resources.

Skill strength for a skill k (SS_k):

$$SS_j = \frac{a_k - a_k^{\min}}{a_j^{\max} - a_j^{\min}} \tag{5.4}$$

The variable a_k describes the number of resources mastering the skill k. It is called skill availability of skill k. Skill availability can be calculated as follows:

$$a_k = \sum_{i \in R} c_{i,k} \tag{5.5}$$

The binary variable $c_{i,k}$ equals 1 if resource i masters skill k and is equal to 0 otherwise.

$$c_{i,k} = \begin{cases} 1, & \text{if } k \in K_i \\ 0, & \text{otherwise} \end{cases} \tag{5.6}$$

K is the set of skills and $r_{j,k}$ is the amount of resources mastering skill k an activity j needs for processing.

The *minimum skill availability* $a_k^{\min}$ signifies the least quantity of a particular skill necessary for the feasible execution of the entire project:

$$a_k^{\min} = \max_{j \in A} r_{j,k} \tag{5.7}$$

A is the set of all activities.

The *maximum skill availability* $a_k^{\max}$ is ascertained by calculating the total skill requirements for a particular skill across all activities:

$$a_k^{\max} = \sum_{j \in A} (r_{j,k}) \tag{5.8}$$

Skill strength for the project (SS):

$$SS = \frac{\sum\limits_{k \in K} a_k - \sum\limits_{k \in K} a_k^{\min}}{\sum\limits_{k \in K} a_k^{\max} - \sum\limits_{k \in K} a_k^{\min}} \tag{5.9}$$

- *Resource availability*: The resource availability RA metric is a measure of resource scarcity. It quantifies the number of resources available for each activity, considering their skills and availability during the project's duration. Furthermore, it assesses the extent of multi-skilled resources and the distribution of skills across the available resources.

  Resource availability (RA):

$$RA = \frac{|R| - R^{\min}}{R^{\max} - R^{\min}} \tag{5.10}$$

The RA metric is determined by dividing the total number of resources, denoted as $|R|$, by the maximum number of resources required for project execution, represented by $R^{\max}$. To account for the minimum number of resources needed to form a feasible workforce, $R^{\min}$ is considered when adjusting these values. This metric facilitates the assessment of overall resource competence and the balance of skill distribution within the workforce, offering crucial insights for efficient project scheduling and resource allocation.

In this thesis, the aforementioned metrics have been chosen for instance selection in the context of the experiments. Other metrics exist as well. For all the metrics mentioned above, the metrics' variabilities (α) are defined [6].

5.1.2 Instances

Five sets of instances, each with different characteristics, are considered for the experiments, and representative instances from each set are selected for testing:

- MSLIB1 (6600 instances): This basic set focuses on medium-sized resources and emphasises the number of resources and skills mastered. Consist from instances with 30 project activities. Project activities are all activities without the two dummy two activities for project start and project end. It is suitable for testing algorithm performance on moderately complex projects.

- MSLIB2 (9000 instances): The set targets projects of greater complexity and higher challenges for the algorithm, requiring more

skills and resources. Consists of projects with 30, 60 and 90 project activities.

- MSLIB3 (11880 instances): This set features many instances and appears to emphasise the distribution of skills across resources. It provides a diverse set of instances to test algorithm performance across various scenarios. All instances in this set have 30 project activities.

- MSLIB4 (5000 instances): This set contains the most challenging instances for projects with 30 activities, offering a rigorous test of algorithm effectiveness on smaller yet highly complex projects. Consist of projects with 30 project activities.

- MSLIB5 (19 instances): This set comprises empirical instances derived from real-world projects, providing an opportunity to evaluate the algorithm's performance in practical situations. This set has instances with various numbers of project activities ranging from 13 to 95.

For the experiments, some instances were selected from MSLIB2. The first choice was MSLIB5, but the preliminary tests showed that its instances were too easy and were mostly solved during the initialisation of the food sources. Similarly, many instances from MSLIB1 didn't show any conclusive results to speak about the algorithm's quality. So, the MSLIB2 was chosen instead. MSLIB3 and MSLIB4 were used for parameter tuning, which will be discussed later in Section 5.2.2. This set contains projects of different sizes (30, 60 and 90 project activities). All sizes have the same number of selected instances. For each size, 32 instances were selected, so their SF, SS and RA metrics vary and span through the possible values. For the SF metric, instances with values 0.25, 0.5, 0.75 and 1 were selected. For the SS, only values 0.3 and 0.7 were considered for the experiments. And finally, for the RA, values 0, 0.4, 0.6 and 1 were chosen. With these values, all possible combinations have their representative instance selected for the experiments. This results in a total of 96 instances from this set.

5.2 Parameter tuning

This section discusses the parameters of the implementation of ABC and their setting. First, a summary of all parameters is introduced. Parameters are of two categories. One category is the parameters for stopping criteria. These include the maximum number of iterations and the maximum number of iterations without improvement in a row. The second category includes parameters selected for tuning, which are discussed in more detail. The tuning procedure and its results follow.

5.2.1 Stopping criteria parameters

The implementation has two criteria for stopping. The algorithm is terminated when either is reached. The first is the maximum number of iterations. It works as an overall ceiling for the algorithm and is set to 1000 iterations. This criterion was never reached from the algorithm runs for the parameter tuning. It exists for potentially bigger problems, which could take too long to solve. The second criterion, the maximum number of iterations in a row without the best food source improvement, is the more important criterion. It stops the algorithm when it platoons on a food source and does not improve any more. Each iteration which fails to improve the best food source increments a counter for this criteria. If iteration produces a better food source than the best one, the counter resets to zero. This parameter was set to 50 iterations without improvement in a row. The preliminary tests have shown that the good enough makespan is reached and is no longer improved. It is also confirmed by findings at **Figure 5.1**–**5.3** where the convergence is clear.

5.2.2 Tuned parameters

Unlike the stopping criteria parameters, the tuned parameters impact the algorithm run itself in terms of convergence speed and food source quality. These parameters are the number of food sources and the trial limit, the number of times a food source can fail to improve until it is abandoned and a new one is initialised instead.

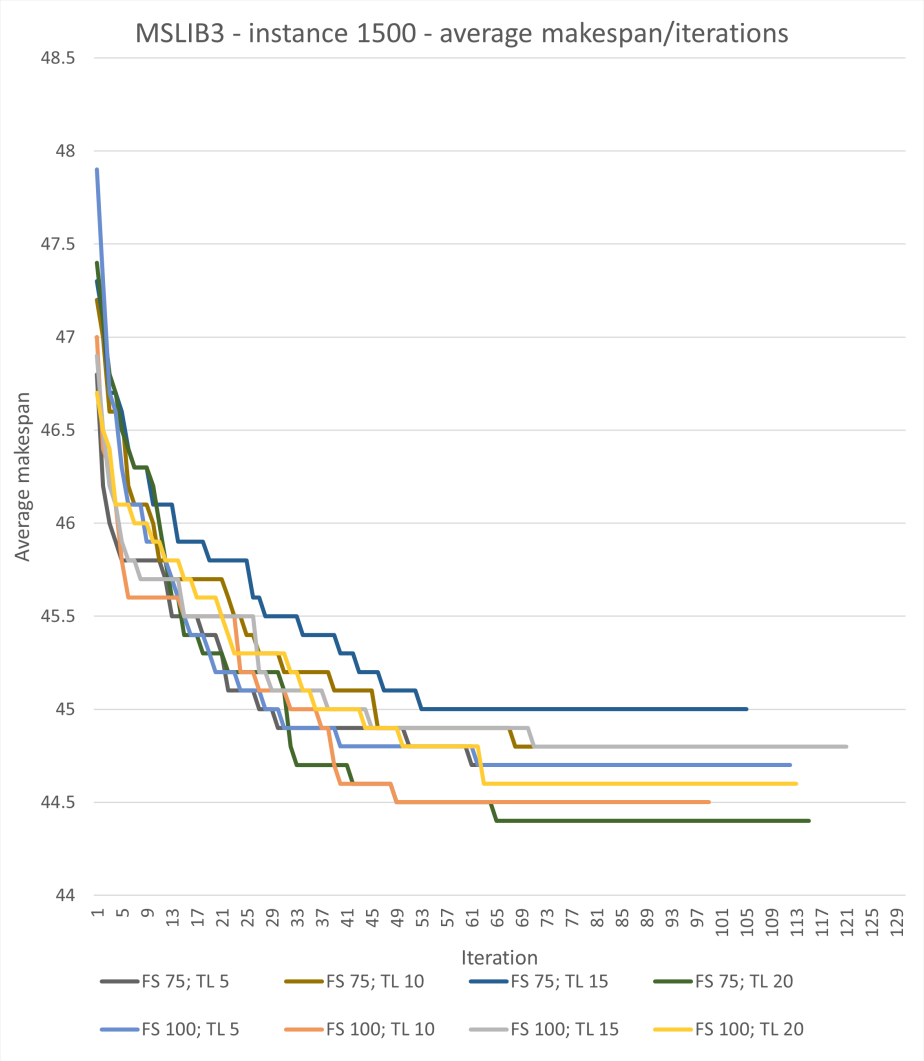

Figure 5.1: Makespan/iterations for instance 1500 from MSLIB3. FS is short for food source count, and TL means trial limit

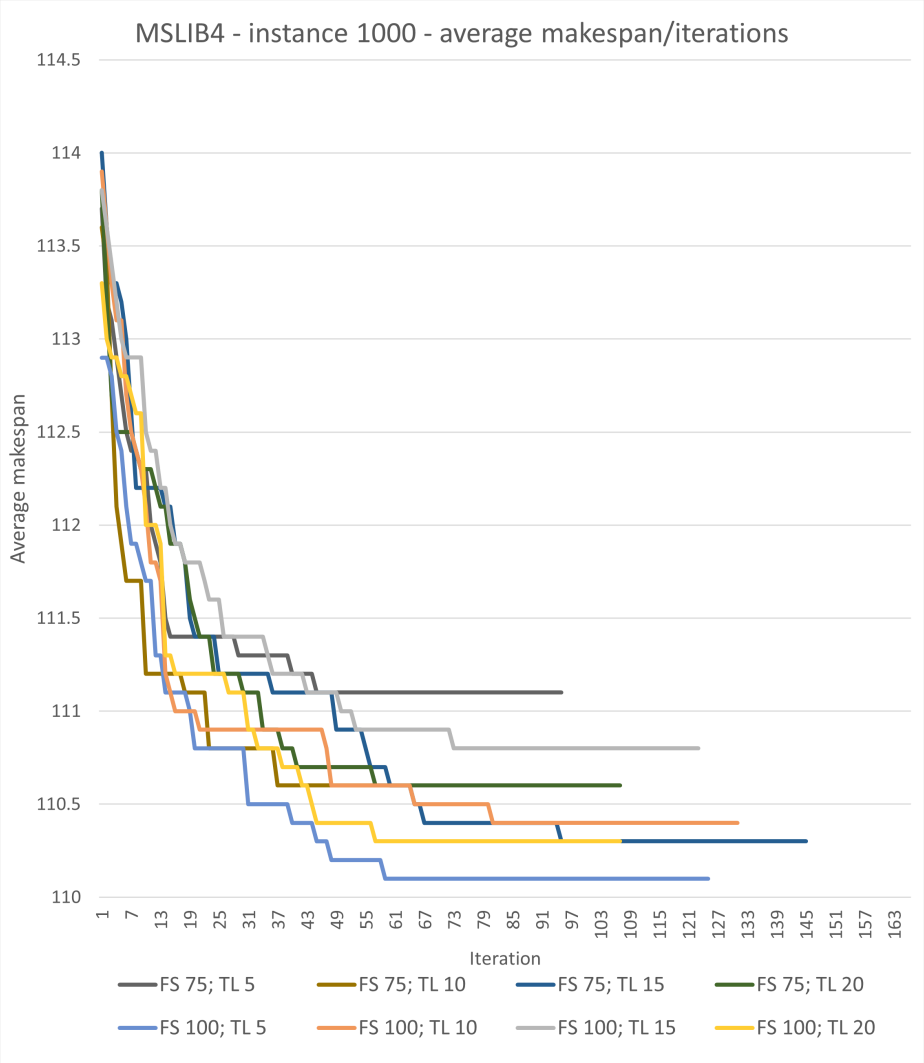

Figure 5.2: Makespan/iterations for instance 1000 from MSLIB4. FS is short for food source count, and TL means trial limit

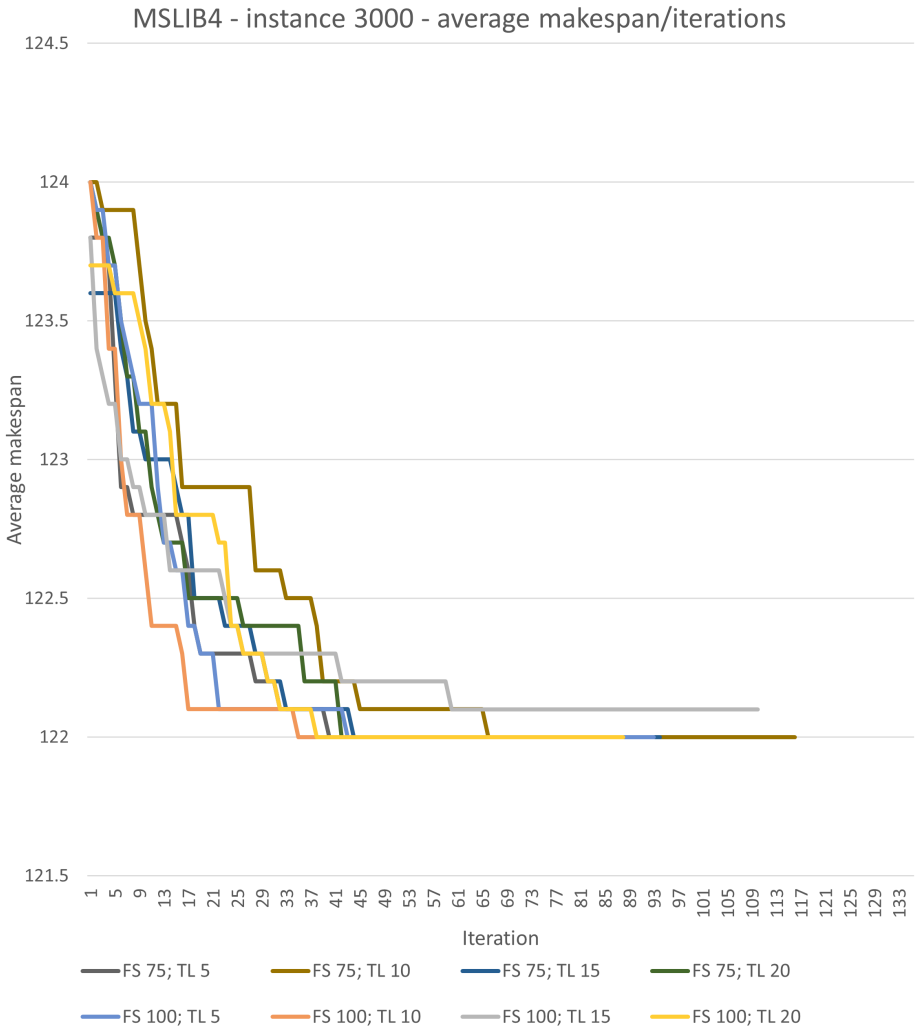

Figure 5.3: Makespan/iterations for instance 3000 from MSLIB4. FS is short for food source count, and TL means trial limit

For the tuning, three from two sets were chosen for a total of six instances. Namely, instances 1500, 4500 and 7500 from MSLIB3 and 1000, 3000 and 5000 from MSLIB4 were selected. The tuning is realised with the grid search. A few candidates for both the number of food sources and trial limit parameters were chosen for testing. Each combination of the parameters was run ten times to obtain reliable values. The averages were considered for the tuning itself.

The results show that the solution's quality and the number of iterations are the same for three of these six instances in all runs. These are instances 4500 and 7500 from the MSLIB3 and 5000 from the MSLIB4. They terminated all runs after 50 iterations, meaning the initialisation produced the best food source. And thus were removed from the parameter tuning analysis because they do not provide insight into the quality of convergence of the algorithm implementation. The rest of the instances for parameter tuning are described in **Table 5.1**. The table contains the basic metrics described in Section 5.1.1 and the number of resources **# R**. Furthermore, J. Snauwaert and M. Vanhoucke's study [6] provides the upper bound makespan **UB**, which represents the best quality solution obtained, along with the corresponding **CPU** processing time used for its discovery. This information can be found on the website [18]. All instances have four types of skills.

Table 5.1: Instances selected for parameter tuning

Set	Instance	SF	SS	# R	RA	UB	CPU
MSLIB3	1500	0.75	0.5	81	0.21	40	52.53
MSLIB4	1000	0.50	0.7	78	0.59	18	21.92
MSLIB4	3000	1.00	0.1	43	1.00	122	35.31

The results from all three tuning instances are shown in **Figure 5.4**–**5.6**. Both the average makespan and CPU from all runs are shown.

Number of food sources

The candidates for the number of food sources were 25, 50, 75 and 100. Generally, the higher the number of food sources, the lower the makespan. Depends on the tested instance, but it can be assumed once a sufficient number of food sources is reached, on average, the

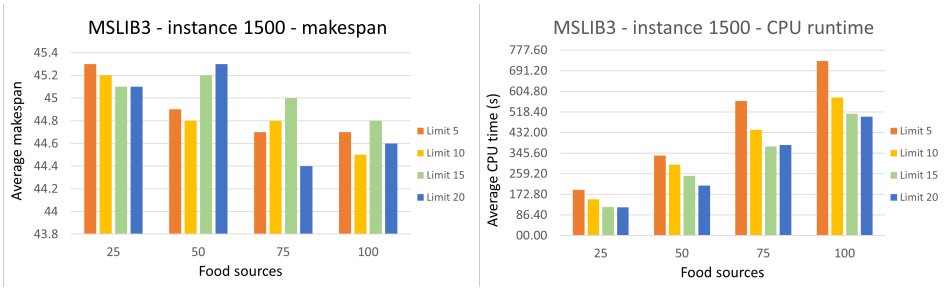

Figure 5.4: Makespan and CPU for instance 1500 from MSLIB3

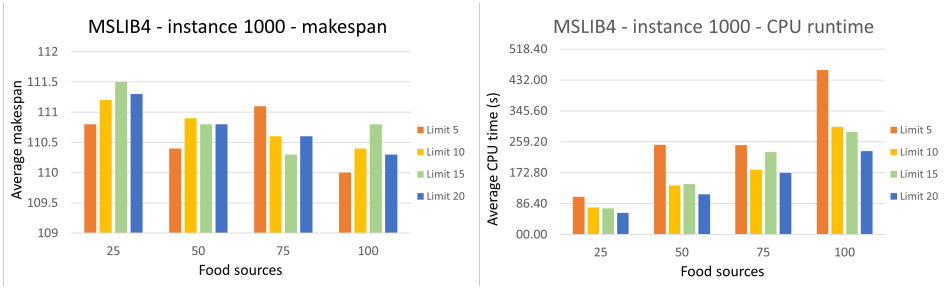

Figure 5.5: Makespan and CPU for instance 1000 from MSLIB4

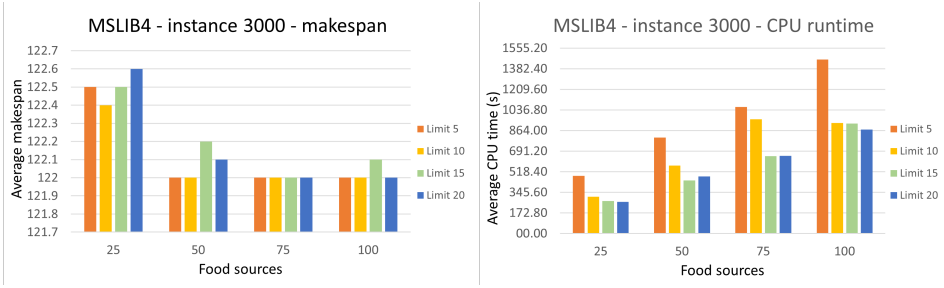

Figure 5.6: Makespan and CPU for instance 3000 from MSLIB4

makespan does not improve anymore. On the other hand, the higher the number of food sources, the higher the CPU time. The final selection for the parameter thus depends on the makespan-CPU time trade-off. In conclusion, for experiments, the value for the number of food sources is set at 100 since time is not a critical factor.

The **Figure 5.1–5.3** show the improvement of the makespan over iterations for each of the three selected instances.

Limit

The limit parameter was tested for values 5, 10, 15 and 20. From the perspective of makespan, the limit parameter does not play a critical role. On the other hand, for the limit 5, the CPU time goes up significantly, as can be seen in the **Figure 5.4–5.6**. But the higher values all yield similar results. In conclusion, from these instances, the most likely candidate for the best limit is 10. On average, it yields excellent results, and the CPU time is much better than the limit set to 5.

5.3 Results

This section contains the results of the experiments. The process of instance selection for the experiments was described in Section 5.1.2. Each instance run was repeated 10 times. **Table 5.3–5.5** summarises the information about the instances and the computational results. For figures comparing the best runs, please refer to Appendix B, **Figure B.1–B.3**. They contain the instance number corresponding to the number in the file on the website [18]. Then the tables contain three chunks of data. The first chunk **Instance metrics** provides basic metrics, described in Section 5.1.1 and the number of resources **# R**. All instances have 4 skills. The second chunk **ABC results** contains the information from the run of our ABC algorithm. The first field in this chunk is the best result from runs of ABC implementation. The best of the ten runs was selected first based on the makespan **UB** and secondary based on the CPU time **CPU** in seconds. Other fields contain information about the average makespan $\overline{\textbf{MS}}$, the average CPU time $\overline{\textbf{CPU}}$ and the variance of the makespan σ^2 **MS**. The next fields contain information about the makespan of the initial solution, average initial makespan $\overline{\textbf{MS}_0}$ as

Table 5.2: Instances' columns description

Instance metrics	Name	Name identification of the instance
	SF	Skill factor of the instance
	SS	Skill strength of the instance
	# R	Number of resources of the instance
	RA	Resource availability of the instance
ABC results	UB	Makespan upper bound found
	CPU	CPU time used for the solution found
	$\overline{\text{MS}}$	Average makespan from all runs
	σ^2 MS	Variance of the makespan
	$\overline{\text{CPU}}$	Average CPU time used for solution
	$\overline{\text{MS}_0}$	Average initial makespan after initialisation
	σ^2 MS_0	Variance of the initial makespan
	NoI	Number of iterations run
Results in [6]	UB	Upper bound of the instance
	CPU	CPU time used for the solution

well as the variance of the makespan after the initialisation $\sigma^2\overline{\text{MS}_0}$. In addition, the number of iterations **NoI** of the best run is shown for the context. Then as the last chunk of information **Results in [6]**, the tables contain the best solution from J. Snauwaert and M. Vanhoucke [6] for comparison. Namely, the best makespan **UB** and the processing time **CPU** in seconds are displayed in the tables. All instances have four types of skills. The **Table 5.2** describes the tables' columns.

The results show that in 43 out of 96 instances, the ABC managed to find the best known solution. In 74 out of the 96 instances, the makespan found by the ABC is less than 10% higher than that of [6]. Similarly, in 90 out of 96 instances, makespan is less than 20% higher. On average, the solution produced by the ABC algorithm is about 6% worse in terms of a makespan than the best one found in [6].

Especially in **Table 5.4**, it can be seen that while the makespan is often higher than the one achieved by [6], the variance of the makespan equals 0 in 31 out of 32 instances. This means the ABC consistently fails to find a better solution in its neighbourhood. Implementing more operators would introduce larger neighbourhoods which might lead to a better-quality solution. This outlines some limitations of this work while indicating future research possibilities.

Table 5.3: MSLIB2 – instances with 30 activities. The numbers in bold are the best makespans found for the instance

Instance metrics					ABC results								Results in [6]	
Name	SF	SS	# R	RA	UB	CPU	$\overline{MS}$	σ^2 MS	$\overline{CPU}$	$\overline{MS_0}$	σ^2 MS_0	NoI	UB	CPU
34	0.25	0.3	10	0.0	36	74.25	37.4	0.49	63.71	39.3	0.90	75	**35**	9.55
1844	0.25	0.3	18	0.4	**122**	9.39	122.0	0.00	10.14	122.0	0.00	50	**122**	10.64
2450	0.25	0.3	24	0.6	**146**	14.28	146.0	0.00	15.19	146.0	0.00	50	**146**	11.10
1256	0.25	0.3	36	1.0	**95**	53.09	95.0	0.00	55.48	95.0	0.00	50	**95**	13.26
1291	0.25	0.7	25	0.0	**93**	18.89	93.0	0.00	19.86	93.0	0.00	50	**93**	11.90
1902	0.25	0.7	33	0.4	**119**	17.52	119.0	0.00	18.65	119.0	0.00	50	**119**	12.27
1910	0.25	0.7	42	0.6	**115**	18.98	115.0	0.00	19.74	115.0	0.00	50	**115**	12.65
1917	0.25	0.7	50	1.0	**120**	19.02	120.0	0.00	19.85	120.0	0.00	50	**120**	12.77
182	0.50	0.3	16	0.0	53	165.67	54.1	0.54	170.06	56.1	0.99	71	**52**	16.40
192	0.50	0.3	35	0.4	29	177.58	29.9	0.32	211.62	32.1	0.54	67	**25**	20.04
200	0.50	0.3	39	0.6	26	114.05	26.6	0.49	149.45	29.4	0.49	58	**23**	18.04
206	0.50	0.3	58	1.0	25	252.47	25.1	0.10	309.72	27.9	0.77	73	**22**	23.69
241	0.50	0.7	44	0.0	32	83.55	32.0	0.00	87.08	32.6	0.49	51	**25**	16.59
253	0.50	0.7	70	0.4	29	94.36	29.0	0.00	98.24	29.0	0.00	50	**25**	20.47
256	0.50	0.7	83	0.6	**14**	94.48	14.0	0.00	98.00	14.0	0.00	50	**14**	21.09
267	0.50	0.7	112	1.0	34	73.56	34.0	0.00	78.39	34.0	0.00	50	**26**	22.60
931	0.75	0.3	22	0.0	**73**	173.36	73.7	0.23	140.79	76.1	0.99	114	**73**	17.64
941	0.75	0.3	45	0.4	64	58.50	64.0	0.00	60.74	64.2	0.40	50	**62**	21.32
948	0.75	0.3	59	0.6	56	87.56	56.0	0.00	92.48	56.0	0.00	50	**50**	26.21
956	0.75	0.3	73	1.0	47	110.87	47.0	0.00	114.38	47.0	0.00	50	**37**	23.69
993	0.75	0.7	49	0.0	**74**	53.34	74.0	0.00	55.14	74.0	0.00	50	**74**	21.07
1001	0.75	0.7	99	0.4	**57**	90.83	57.0	0.00	92.81	57.0	0.00	50	**57**	33.70
1006	0.75	0.7	122	0.6	**37**	94.22	37.0	0.00	100.30	37.0	0.00	50	**37**	33.15
1016	0.75	0.7	168	1.0	75	113.87	75.0	0.00	120.08	75.0	0.00	50	**65**	33.73
1081	1.00	0.3	27	0.0	67	410.20	70.2	1.73	243.34	75.7	0.68	143	**66**	29.75
1092	1.00	0.3	60	0.4	**49**	214.34	49.8	0.84	196.41	51.6	0.27	91	**49**	37.04
1096	1.00	0.3	76	0.6	68	227.06	68.9	0.32	214.88	71.6	2.04	71	**66**	39.46
1106	1.00	0.3	94	1.0	**43**	125.44	43.0	0.00	129.33	43.0	0.00	50	**43**	28.85
1141	1.00	0.7	59	0.0	44	99.91	44.0	0.00	123.48	45.4	1.38	51	**43**	34.47
1151	1.00	0.7	128	0.4	**46**	106.41	46.0	0.00	109.51	46.0	0.00	50	**46**	45.58
1156	1.00	0.7	157	0.6	**61**	169.30	61.0	0.00	173.22	61.0	0.00	50	**61**	49.53
1166	1.00	0.7	205	1.0	**59**	176.16	59.0	0.00	187.59	59.0	0.00	50	**59**	46.08

Table 5.4: MSLIB2 – instances with 60 activities. The numbers in bold are the best makespans found for the instance

Instance metrics					ABC results								Results in [6]	
Name	SF	SS	# R	RA	UB	CPU	$\overline{MS}$	σ^2 MS	$\overline{CPU}$	$\overline{MS_0}$	σ^2 MS_0	Nol	UB	CPU
3031	0.25	0.3	18	0.0	61	228.60	61.7	0.46	215.60	64.4	1.60	86	**53**	48.57
3645	0.25	0.3	35	0.4	101	46.20	101.0	0.00	47.26	101.0	0.00	50	**93**	57.96
3648	0.25	0.3	45	0.6	**99**	58.20	99.0	0.00	59.66	99.0	0.00	50	**99**	59.97
3656	0.25	0.3	56	1.0	100	59.00	100.0	0.00	59.72	100.0	0.00	50	**91**	57.24
3691	0.25	0.7	53	0.0	**102**	54.60	102.0	0.00	56.16	102.0	0.00	50	**102**	57.64
3701	0.25	0.7	82	0.4	133	70.30	133.0	0.00	71.88	133.0	0.00	50	**129**	64.91
3709	0.25	0.7	89	0.6	104	72.80	104.0	0.00	76.84	104.0	0.00	50	**98**	66.06
3716	0.25	0.7	106	1.0	116	63.10	116.0	0.00	65.97	116.0	0.00	50	**113**	62.85
3781	0.50	0.3	34	0.0	110	83.50	110.0	0.00	87.52	110.2	0.40	50	**104**	74.22
3792	0.50	0.3	63	0.4	90	114.40	90.0	0.00	116.94	90.0	0.00	50	**88**	76.76
3796	0.50	0.3	79	0.6	94	145.80	94.0	0.00	151.79	94.0	0.00	50	**93**	80.34
3806	0.50	0.3	101	1.0	102	138.70	102.0	0.00	142.83	102.2	0.40	50	**94**	77.14
3841	0.50	0.7	85	0.0	**105**	103.80	105.0	0.00	107.15	105.0	0.00	50	**105**	79.06
3851	0.50	0.7	123	0.4	93	129.70	93.0	0.00	134.68	93.0	0.00	50	**86**	81.63
3856	0.50	0.7	146	0.6	**105**	141.30	105.0	0.00	147.47	105.0	0.00	50	**105**	86.10
3866	0.50	0.7	213	1.0	**106**	152.20	106.0	0.00	153.69	106.0	0.00	50	**106**	91.28
3931	0.75	0.3	56	0.0	**110**	113.60	110.0	0.00	116.90	110.0	0.00	50	**110**	84.37
3941	0.75	0.3	86	0.4	**108**	166.20	108.0	0.00	171.62	108.0	0.00	50	**108**	92.69
3946	0.75	0.3	101	0.6	**91**	172.50	91.0	0.00	190.48	91.4	0.27	51	**91**	90.29
3956	0.75	0.3	147	1.0	87	271.60	87.0	0.00	275.36	87.0	0.00	50	**84**	93.02
3991	0.75	0.7	99	0.0	101	185.80	101.0	0.00	190.63	101.0	0.00	50	**98**	91.82
4001	0.75	0.7	190	0.4	**114**	182.40	114.0	0.00	189.27	114.0	0.00	50	**114**	104.18
4006	0.75	0.7	230	0.6	112	231.00	112.0	0.00	237.78	112.0	0.00	50	**107**	114.50
4016	0.75	0.7	317	1.0	106	264.30	106.0	0.00	272.08	106.0	0.00	50	**102**	149.77
4081	1.00	0.3	49	0.0	120	183.20	120.0	0.00	240.30	120.5	0.28	50	**110**	67.89
4091	1.00	0.3	108	0.4	**127**	222.10	127.0	0.00	247.14	127.9	0.54	50	**127**	95.25
4096	1.00	0.3	139	0.6	**87**	314.30	87.0	0.00	495.93	89.3	0.68	56	**87**	99.88
4106	1.00	0.3	181	1.0	116	222.90	116.0	0.00	330.66	117.9	0.99	50	**113**	108.69
4141	1.00	0.7	122	0.0	**112**	173.10	112.0	0.00	182.55	112.0	0.00	50	**112**	86.48
4151	1.00	0.7	240	0.4	84	291.70	84.0	0.00	305.08	84.0	0.00	50	**79**	121.37
4156	1.00	0.7	298	0.6	**118**	373.00	118.0	0.00	391.30	118.0	0.00	50	**118**	145.22
4166	1.00	0.7	416	1.0	88	620.30	88.0	0.00	643.82	88.0	0.00	50	**87**	193.16

Table 5.5: MSLIB2 – instances with 90 activities. The numbers in bold are the best makespans found for the instance

Instance metrics					ABC results								Results in [6]	
Name	SF	SS	# R	RA	UB	CPU	$\overline{MS}$	σ^2 MS	$\overline{CPU}$	$\overline{MS_0}$	σ^2 MS_0	Nol	UB	CPU
6631	0.25	0.3	28	0.0	**126**	92.09	126.0	0.00	100.23	126.4	0.49	50	**126**	126.19
6641	0.25	0.3	48	0.4	**155**	87.77	155.0	0.00	89.33	155.0	0.00	50	**155**	146.93
6649	0.25	0.3	55	0.6	**137**	90.39	137.0	0.00	95.23	137.0	0.00	50	**137**	149.61
6657	0.25	0.3	80	1.0	**159**	106.52	159.0	0.00	110.54	159.0	0.00	50	**159**	158.82
6691	0.25	0.7	73	0.0	**158**	111.03	158.0	0.00	114.02	158.0	0.00	50	**158**	150.08
6701	0.25	0.7	84	0.4	167	93.00	167.0	0.00	97.46	167.0	0.00	50	**161**	153.68
6706	0.25	0.7	115	0.6	**144**	125.58	144.0	0.00	127.35	144.0	0.00	50	**144**	173.88
6716	0.25	0.7	161	1.0	**165**	140.05	165.0	0.00	145.92	165.0	0.00	50	**165**	169.65
6781	0.50	0.3	56	0.0	**156**	529.30	157.5	0.50	299.53	159.9	2.54	114	**156**	171.73
6791	0.50	0.3	98	0.4	**187**	173.77	187.0	0.00	178.55	187.0	0.00	50	**187**	185.11
6796	0.50	0.3	104	0.6	**119**	204.14	119.0	0.00	211.38	119.0	0.00	50	**119**	184.06
6806	0.50	0.3	153	1.0	190	208.62	190.0	0.00	216.23	190.0	0.00	50	**183**	198.94
6841	0.50	0.7	85	0.0	**165**	156.73	165.0	0.00	162.74	165.0	0.00	50	**165**	176.90
6251	0.50	0.7	200	0.4	66	424.86	66.0	0.00	436.45	66.0	0.00	50	**62**	179.72
6256	0.50	0.7	236	0.6	69	414.47	69.0	0.00	441.94	69.0	0.00	50	**65**	196.67
6266	0.50	0.7	323	1.0	**57**	514.72	57.0	0.00	529.26	57.0	0.00	50	**57**	205.68
6331	0.75	0.3	82	0.0	74	437.25	75.5	1.61	871.59	79.9	6.54	50	**64**	175.97
6341	0.75	0.3	118	0.4	73	637.31	74.5	1.17	888.77	79.7	2.68	63	**69**	163.34
6346	0.75	0.3	159	0.6	64	3029.34	66.6	1.82	1607.25	71.2	1.96	156	**57**	215.78
6356	0.75	0.3	206	1.0	65	616.84	65.2	0.18	874.38	67.8	2.62	50	**58**	191.75
6391	0.75	0.7	139	0.0	74	1292.87	76.6	1.38	925.27	79.5	2.06	99	**68**	174.44
6401	0.75	0.7	270	0.4	74	592.55	74.0	0.00	611.79	74.0	0.00	50	**70**	242.83
6406	0.75	0.7	327	0.6	71	801.33	71.0	0.00	847.92	71.1	0.10	50	**61**	273.15
6416	0.75	0.7	487	1.0	72	794.84	72.0	0.00	818.72	72.0	0.00	50	**59**	311.85
6481	1.00	0.3	79	0.0	91	757.47	92.6	1.38	1255.55	97.2	3.73	58	**77**	220.51
6491	1.00	0.3	163	0.4	65	1404.58	66.4	0.49	1106.50	69.2	2.62	89	**51**	214.92
6496	1.00	0.3	197	0.6	68	2397.72	70.3	2.01	1556.93	74.1	1.66	117	**61**	248.34
6506	1.00	0.3	279	1.0	62	3399.28	63.3	0.68	2427.77	67.8	2.84	133	**56**	249.99
6541	1.00	0.7	174	0.0	63	661.08	63.0	0.00	903.59	65.7	3.12	50	**56**	216.04
6551	1.00	0.7	372	0.4	**61**	744.89	61.0	0.00	1172.86	62.7	0.46	51	**61**	271.72
6556	1.00	0.7	433	0.6	83	962.66	83.0	0.00	1031.22	83.0	0.00	50	**65**	325.81
6566	1.00	0.7	608	1.0	69	1532.17	69.0	0.00	1629.68	69.0	0.00	50	**60**	409.26

The average initial food sources' makespans were equal to the average final makespans in 62 out of 96 instances. This was again most notable in the instances with 60 activities, where 24 out of 32 instances yielded this result. This implies that over one-third of the instances were improved by our algorithm. Comparing the initial average makespans with the makespans from [6] gives an interesting view of our results. In 35 out of 96 instances, the average initial makespan equals the makespan of the best solution found in [6]. Overall, this means that simply initialising 100 food sources gets the best-known solution in more than a third of the instances tested for this thesis. In the instances with 30 activities, this was the case for 15 instances. This seems to indicate that the fewer activities, the easier it is to find the best solution just by random initialisation procedure. For instances with 60 and 90 activities, this was the case for 10 instances each.

The CPU runtimes of the best solutions were in 74 out of 96 cases lower than the average runtimes of the instance. The most notable are the instances with 60 activities. Out of the 32 instances in this category, the 31 runtimes of the best solutions were lower than the average runtimes. All best runs of our algorithm except one, instance number 1844, have a higher runtime than the runtimes of the best results from [6]. But due to the lack of CPU specifications used to find the best solutions in their paper, the comparison is questionable.

6 Conclusion

In conclusion, this bachelor's thesis explored the application of the ABC algorithm in the domain of project scheduling, focusing on the multi-skilled resource-constrained project scheduling problem (MSRCPSP). The scheduling problems were discussed with emphasis on the MSRCPSP, which was selected for experiments. Representative solution methods were described, mainly the metaheuristics based on the communication and foraging behaviour of the honey bees. Then, the general idea behind the ABC algorithm was discussed. The implementation itself was introduced next in detail. The benchmarks for the experiments, metrics of the instances and their selection were described. Finally, the parameter tuning and the experiments themselves were discussed.

The results are promising. The makespan of almost half of the instances is the same as that found by the recent work J. Snauwaert and M. Vanhoucke [6] and only a bit higher in most of the rest. However, the CPU time used is higher, but these results are questionable since we do not have information about the CPU used for finding the best solution at [6]. The ABC algorithm shows promising results to be a competitive solution method for the MSRCPSP. To our knowledge, this is the first work applying the ABC on the MSRCPSP, and it is based on cutting-edge research from 2023 and has a great potential for the future. The findings of this research contribute to the advancement of scheduling methodologies and provide insights for the practical implementation of the ABC algorithm in interesting project scheduling scenarios. Further research and enhancements can build upon these findings to continue improving project scheduling techniques using the ABC algorithm.

Implementing other operators would be an exciting way to build upon this work. Operators like swapping eligible activities or resources could help to reach larger neighbourhoods in the search space, thus improving the solution quality. Other ways to build upon this implementation might be modifying and testing multiple acceptance criteria for all bee types or modifying the method of initialising the new food sources in the scout bee phase. For example, the new food source created in the scout bee phase could be the best food source

modified by a particular operator. In the future, a combination of these modifications could be implemented, improving the quality of the solutions of the proposed algorithm.

Another option to continue research is to implement the ABC algorithm for multi-skilled resources with hierarchical skills or to account for the cost of resource skills. An alternative option to work on is a project with variable activity duration. The benchmarks used for this thesis already have these extensions together with some others as modules inside them [18].

Bibliography

1. KARABOGA, Dervis. An Idea Based on Honey Bee Swarm for Numerical Optimization, Technical Report - TR06. *Technical Report, Erciyes University*. 2005.
2. KARABOGA, Dervis; AKAY, Bahriye; OZTURK, Celal. Artificial Bee Colony (ABC) Optimization Algorithm for Training Feed-Forward Neural Networks. In: TORRA, Vicenç; NARUKAWA, Yasuo; YOSHIDA, Yuji (eds.). *Modeling Decisions for Artificial Intelligence*. Springer Berlin Heidelberg, 2007, pp. 318–329.
3. KOLISCH, Rainer; HARTMANN, Sönke. Experimental Investigation of Heuristics for Resource-Constrained Project Scheduling: An Update. *European Journal of Operational Research*. 2006, vol. 174, pp. 23–37.
4. HEGAZY, Tarek; SHABEEB, Abdul Karim; ELBELTAGI, Emad; CHEEMA, Tariq S. ALGORITHM FOR SCHEDULING WITH MULTISKILLED CONSTRAINED RESOURCES. *Journal of Construction Engineering and Management-asce*. 2000, vol. 126, pp. 414–421.
5. HARTMANN, Sönke; BRISKORN, Dirk. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*. 2022, vol. 297, no. 1, pp. 1–14.
6. SNAUWAERT, Jakob; VANHOUCKE, Mario. A classification and new benchmark instances for the multi-skilled resource-constrained project scheduling problem. *European Journal of Operational Research*. 2023, vol. 307, no. 1, pp. 1–19.
7. AFSHAR-NADJAFI, Behrouz. Multi-skilling in scheduling problems: A review on models, methods and applications. *Computers & Industrial Engineering*. 2021, vol. 151, p. 107004.
8. TALBI, El-Ghazali. *Metaheuristics: From Design to Implementation*. John Wiley & Sons, 2009.

9. LAMBORA, Annu; GUPTA, Kunal; CHOPRA, Kriti. Genetic Algorithm- A Literature Review. In: *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing* (*COMITCon*). IEEE, 2019, pp. 380–384.

10. DORIGO, Marco; BIRATTARI, Mauro; STUTZLE, Thomas. Ant colony optimization. *IEEE Computational Intelligence Magazine*. 2006, vol. 1, no. 4, pp. 28–39.

11. YANG, Chunming; SIMON, D. A new particle swarm optimization technique. In: *18th International Conference on Systems Engineering* (*ICSEng'05*). IEEE, 2005, pp. 164–169.

12. RAJASEKHAR, Anguluri; LYNN, Nandar; DAS, Swagatam; SUGANTHAN, P.N. Computing with the collective intelligence of honey bees – A survey. *Swarm and Evolutionary Computation*. 2017, vol. 32, pp. 25–48.

13. SATO, T.; HAGIWARA, M. Bee System: finding solution by a concentrated search. In: *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*. IEEE, 1997, vol. 4, pp. 3954–3959.

14. REZA, Akbari; ZEIGHAMI, Vahid; ZIARATI, Koorush. Artificial Bee colony for resource constrained project scheduling problem. *International Journal of Industrial Engineering Computations*. 2011, vol. 2, pp. 45–60.

15. NOURI, Nouha; KRICHEN, Saoussen; LADHARI, Talel; FATIMAH, Princess. A discrete artificial bee colony algorithm for resource-constrained project scheduling problem. In: *5th International Conference on Modeling, Simulation and Applied Optimization* (*ICMSAO*). IEEE, 2013, 6 pages.

16. ZHONG, Jinghui; HU, Xiaomin; ZHANG, Jun; GU, Min. Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms. In: *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce* (*CIMCA-IAWTIC'06*). IEEE, 2005, vol. 2, pp. 1115–1121.

17. LIN, Jian; ZHU, Lei; GAO, Kaizhou. A genetic programming hyper-heuristic approach for the multi-skill resource constrained project scheduling problem. *Expert Systems with Applications*. 2020, vol. 140, p. 112915.

18. *The multi-skilled resource-constrained project scheduling problem* [`https://www.projectmanagement.ugent.be/research/project_scheduling/MSRCPSP`]. 2023. Accessed: 2023-03-16.

19. CORREIA, Isabel; LOURENÇO, Lídia Lampreia; SALDANHA-DA-GAMA, Francisco. Project scheduling with flexible resources: Formulation and inequalities. *OR Spectrum*. 2012, vol. 34, no. 3, pp. 635–663.

A Information in IS MU

The files inserted into the Information System of Masaryk University:

- thesis.pdf – Text of this thesis in PDF
- README.md – A guidance file for running the experiments with ABC algorithm on the MSLIB instances
- ABC.zip – Contains the implementation of the ABC algorithm for the MSRCPSP. In this folder are:
  - ABC – The folder containing the core of the ABC algorithm
  - ABC.Run – The folder containing the executive part of the algorithm, the configuration file **config.json** for the paths, parameters and selection of the instances
  - ABC.sln – Source code solution of the implementation
  - LICENCE.md – The licence file to the software
- ABC - windows.zip – the self-contained build for windows (also includes its own **config.json**)
- MSLIB.zip – contains benchmarks instances and basic information about the benchmarks as well as the file MSLIB (Parameters and BKS).xlsx with information about the instances and results by [6]
- experiments.xlsx – results of the experiments on the selected instances

B Results in graphs

In this chapter are three figures **Figure B.1**–**B.3** for instances with 30, 60 and 90 activities. These figures show the best run for each instance, comparing the initial makespan, the result of the ABC algorithm and the results from [6].

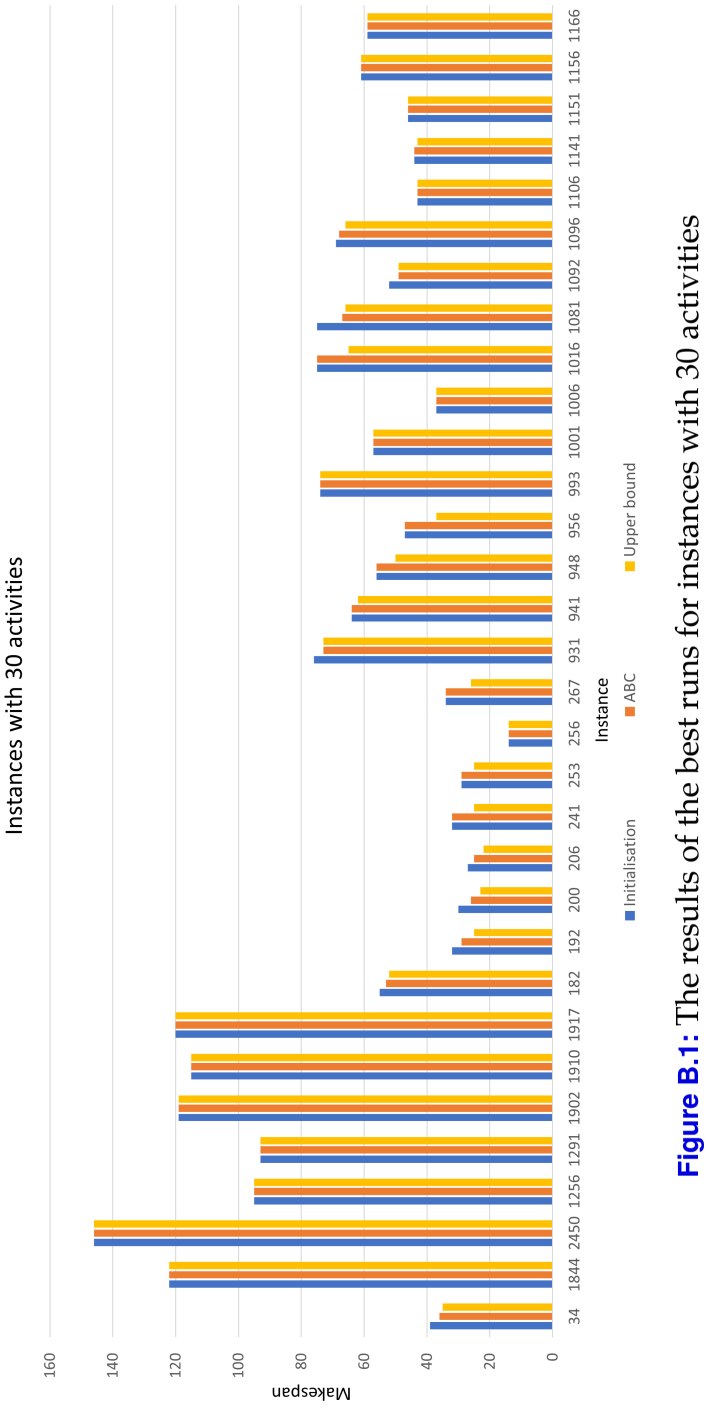

Figure B.1: The results of the best runs for instances with 30 activities

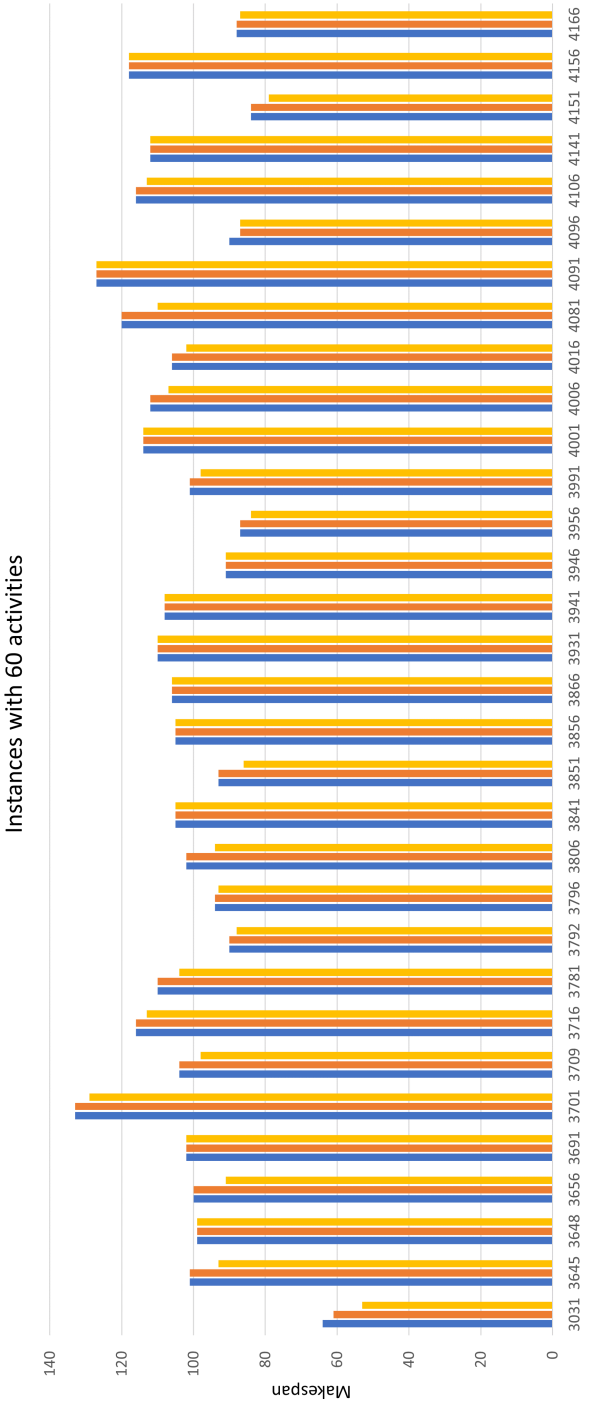

Figure B.2: The results of the best runs for instances with 60 activities

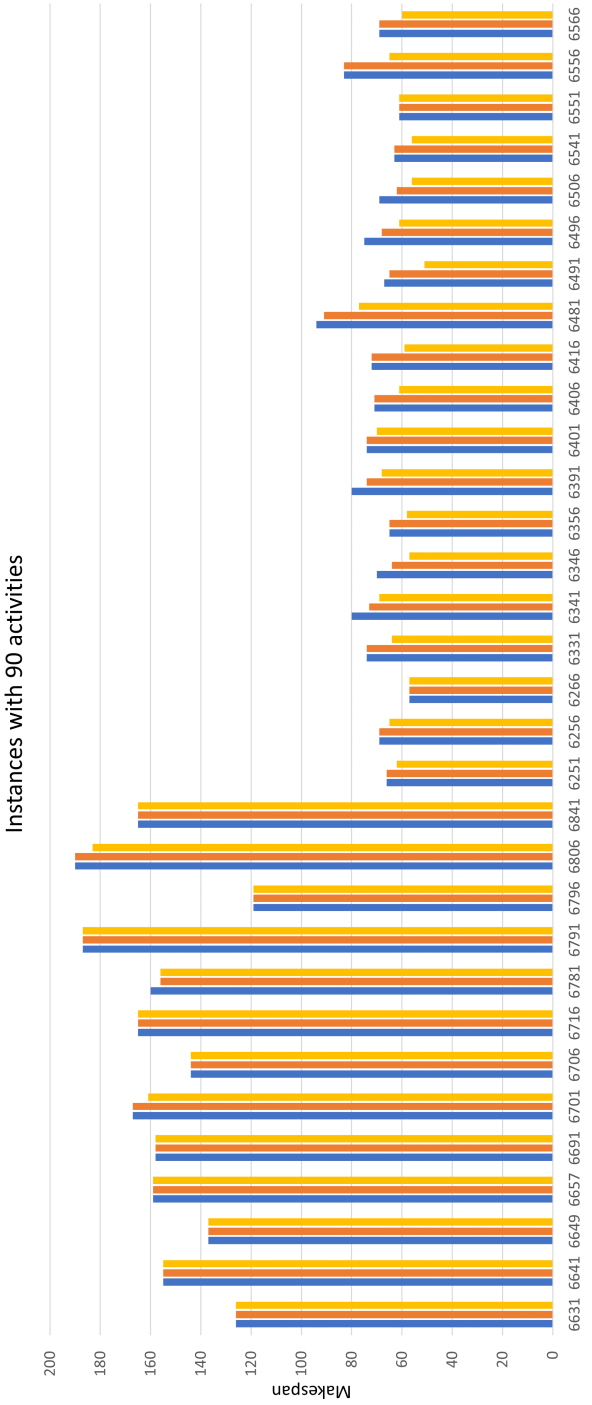

Figure B.3: The results of the best runs for instances with 90 activities