UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS ASSIGNMENT No. 1226

ON-LINE SCHEDULING HEURISTICS IN DISTRIBUTED ENVIRONMENTS

Vlaho Poluta

Zagreb, June 2016

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my master thesis mentor Prof. Domagoj Jakobović, doc.dr.sc. for the continuous support of my research, for his patience, motivation and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

Also, I must express my very profound gratitude to my family and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. Special thanks go to the person who has greatly helped me with proofreading this thesis. This accomplishment would not have been possible without them.

Thank you.

Content

1. Introduction

Scheduling is a decision-making process that is used on a regular basis in many manufacturing and service industries. It deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives. [1]

The resources and tasks can take many different forms. The resources may be machines in a workshop, workers in a company, materials in factories, processing units in a computing environment and so on. The tasks may be operations in a production process, stages in projects, creating parts in factories, executions of computer programs and so on. Each task can have a certain priority level, the earliest possible start time and an expected finish time. The objectives can also take many different forms. One objective may be the minimization of the completion time of the last task and another may be the minimization of the number of tasks completed after their respective due dates. [1]

There might be single or multiple machines in a scheduling system. The single machine environment is simple and a special case of all environments. Single machine environments often have properties that none of the multiple machine environments have, such as the ability to be optimized with basic heuristics. Multiple machines can work in series or in parallel. They often combine serial and parallel execution of tasks.

In computing, scheduling is the method by which work specified by some means is assigned to resources that complete the work. The work may be virtual computation elements such as threads, processes or data flows, which are in turn scheduled onto hardware resources such as processors, network links or expansion cards. [2]

Scheduling, as a decision-making process, plays an important role in most manufacturing and production systems as well as in most information processing environments. It is also important in transportation and distribution settings and in other types of service industries. [1]

A scheduler is what carries out the scheduling activity. Schedulers are often implemented so they keep all the computer resources busy (as in load balancing), allow multiple users to share system resources effectively, or to achieve a target quality of service. Scheduling is fundamental to computation itself, and an intrinsic part of the execution model of a computer system. [2]

This thesis focuses on a specific case of multiple machines scheduling in distributed systems. A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages [3]. In the next chapter, the thesis will first present the specific problem that it tries to solve. Then it will describe known methods that were used in trying to solve this specific problem and afterwards the designed scheduling system will be presented with its results.

2. About the Assignment

The specific scheduling problem that the thesis tries to optimize is presented in this chapter. First, the concrete problem will be presented together with its properties and it will be fitted into scheduling terminology. The second part of this chapter will focus on presenting the abstract ideas of solving the problem. It deals with the three main phases of the thesis as well as the exact part of the scheduling problem that it is trying to solve in each of them. The simplifications that are used in solving their part of the scheduling problem are also presented.

2.1 Introduction to the Assignment

The specific scheduling problem that was tackled in this thesis is rather an unusual one. The problem deals with tasks being scheduled on executing nodes through networks of scheduling servers. The issue is to schedule those tasks in a way to optimize makespan. The makespan is the total length of the schedule (that is, the moment when all the jobs have finished processing). The three major components are tasks, executing nodes and scheduling servers. In this problem, the executing nodes can be thought of as machines since tasks can only be executed on them. Those machines have eligibility restrictions because they can only execute one task at a time. Tasks have precedence constraints because they usually require other tasks to be completed before they can start. Tasks are also machine constrained because every task can be executed only at the exact executing node (machine).

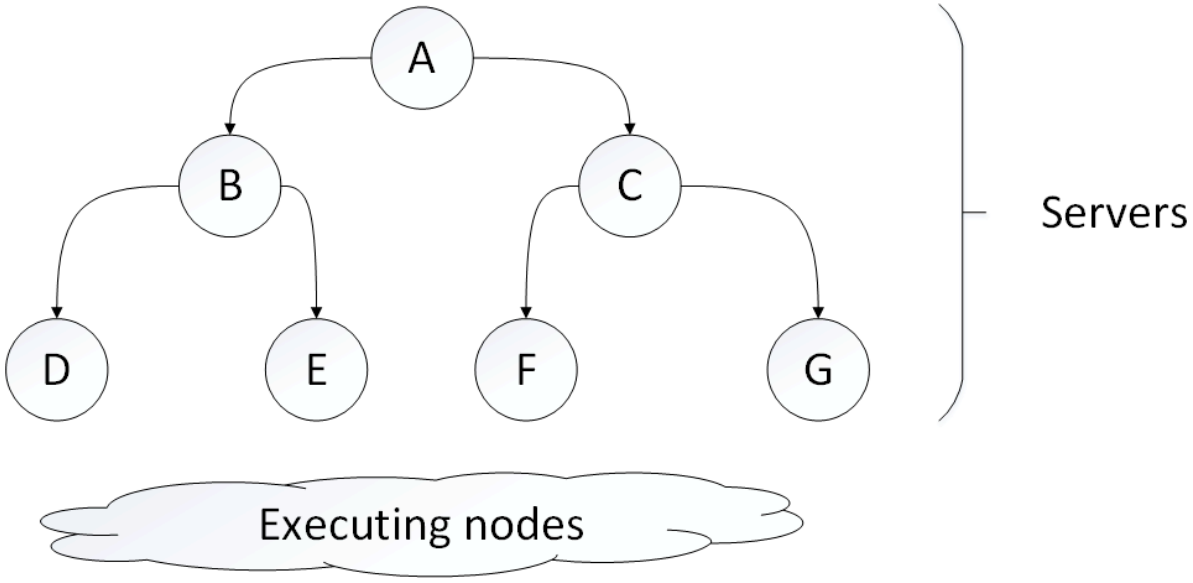

Image 2-1 Distribution of scheduling servers

The description will begin with the most abstract elements; the scheduling servers. Usually, there is a small number of scheduling servers (for example seven) that are connected in a binary tree as seen in Image 2-1. They transfer tasks and information

about them (like their scheduled finish time) as messages through their connections in the binary tree. Any of those scheduling servers can connect to any of the executing nodes, but at any given time only one server can be connected to a specific executing node. The scheduling server needs to be connected to the executing node in order to execute a task on it. The server can have multiple parallel connections.

There are a lot more executing nodes in the system than there are scheduling servers (there is usually around 30,000 executing nodes). This presents a critical limitation since the scheduling servers are bound by their memory capacity and cannot have as many open connections to the executing nodes as they want to (the maximum is usually 40). The scheduling servers have to constantly open and close the connections in order to execute the tasks.

Some tasks have the outcomes of other tasks as preconditions. Such dependencies essentially form a graph. The graphs range from one task with no dependencies to thousands or hundreds of thousands of connected tasks (there are usually three million tasks in the system). One other key insight here is that, if the task T1 is executed on the server B and T2 requires T1 to finish, T2 must wait at root node (A on image 1) for the confirmation that T1 is scheduled. Then the root server can assign task T2 to itself or to some of its subtrees.

The main responsibility here goes to the scheduling servers and their ability to schedule the assigned tasks. Scheduling is done in order to reduce the amount of opening and closing the connections to the executing nodes and to increase parallelism.

Another key feature of this system is that it gets new tasks every hour. This requires the evaluation to be fast and the solution to be some heuristic that can schedule the tasks effectively on-line.

The important system properties for this specific scheduling problem can be found in Table 2-1.

Table 2-1 Important system properties

Property name	Description	Default value
Alpha	Time to open a connection to an executing node	1000 ms
Beta	Time to close a connection to an executing node	200 ms
Gamma	Time to execute a task	50 ms
Delta	Time to send a task one level below in binary tree	10 ms
Epsilon	Time to send task results one level above.	2 ms

2.2 Simplification, Separation of Concerns

Since the initial problem is a big and unusual one, complete testing and experimenting on that problem can be actually separated into three phases. In each phase some assumptions have been made and some simplifications have been done in order to preserve all of the problem's hard constraints.

2.2.1 The First Phase

During the first and the initial phase of the testing, the idea was to create a system that has only one scheduling server. This simplified things a lot since there was no longer the need to send tasks through the server tree and wait for a response from other servers. Now there were only tasks and executing nodes. The system can be represented with Image 2-2.

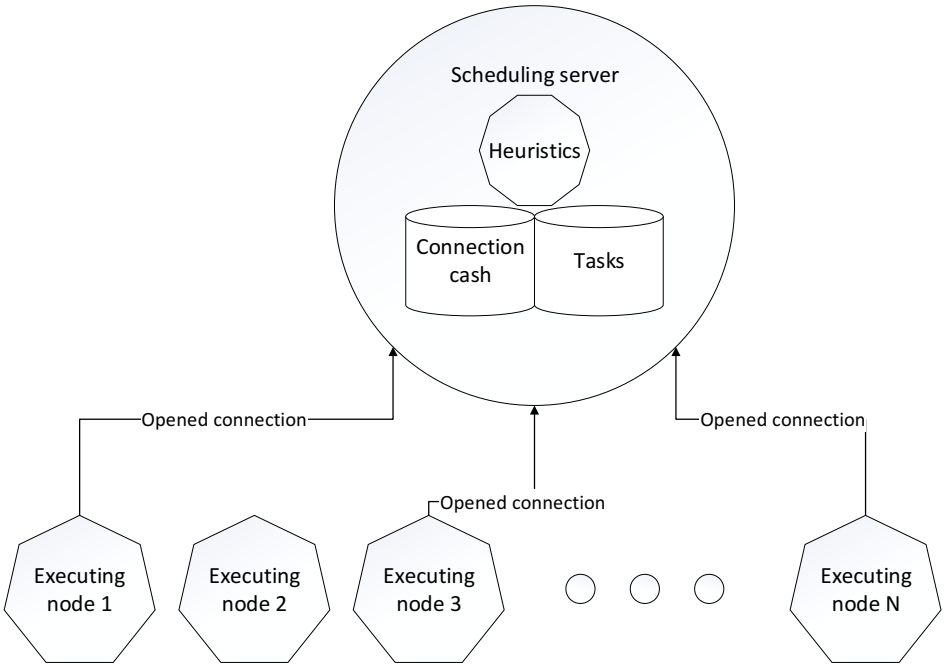

Image 2-2 System with a single scheduling server

Such a simplified problem can then be divided into two subproblems. The first one being to find a heuristic that can act as a comparator for the tasks to be able to determine the best task currently possible to schedule for the execution. The second subproblem is to determine which executing node to disconnect when we need to open a new connection.

This worked fine and produced some results, however this approach was not able to execute in real time, that is, in less than an hour for the load (around three million tasks) that it should be able to process. With this approach it took approximately 70 hours to evaluate the given load. The biggest problem with this approach was that the algorithm was comparing the remaining tasks (to find the best one) in each

iteration. The reason for this is because the priority of the tasks can change dynamically depending on the system state, for example, if the required executing node gets connected or disconnected.

2.2.2 The Second Phase

In the second phase, all the simplifications remained the same. The main idea of this phase was to tackle with one hour limitation. Since there were a lot more tasks (around three million) than executing nodes (around ten thousand), and even less nodes could be connected at the same time (around forty), the idea was to schedule the executing nodes first and then schedule the tasks. As with the tasks, the same approach was needed for the executing nodes. The idea was to find the heuristic that can act as a comparator for the executing nodes and determine the best executing node that the scheduling server can connect to. This was not very effective since in every iteration we had to find the best executing node and then find the best task in it. Not performing much better leads to new approaches that would somewhat loosen the flexibility of the schedule. The new idea was to keep the connections open as long as there are tasks ready for execution. This approach was not that harmful since opening and closing a connection usually lasts much longer than executing a task (1000 ms to open a connection, 200 ms to close it and 50 ms to execute a task).

This approach was a lot faster than the one in the first phase. Using this approach, the scheduler was able to evaluate the task load (around three million) in 20-40 minutes, depending on the implementation and the heuristics that was used. In this phase, there are now two optimization points; comparing the executing nodes and comparing the tasks.

2.2.3 The Third Phase

One server simplification was removed in this phase. The main idea was to use the best heuristics (comparators) from the previous phase and to build on them. Now the focus turns to the scheduling servers and their communication. To be more exact, the main focus of this phase is to create a heuristic that the scheduling server can use in order to be able to assign tasks to itself or any of its subtrees. Since assigning a task by task can also take a lot of time, the objective was to assign a node by node and then group the tasks according to their executing nodes. Since the tasks that have prerequisite tasks cannot be moved from the root server until their prerequisites are done, sending new tasks could also represent a problem.

3. Techniques Used

Since there are a lot of tasks and executing servers, the heuristic that would be able to find the best task or executing node must act as a comparator for tasks or executing nodes. For the initial tests, a lot of hand written heuristics were used for testing the evaluator and later on for testing the heuristics that are learned by some machine learning techniques. The machine learning techniques that were used are described in more detail further in this chapter. But before going into machine learning techniques or even those hand written heuristics, it is important to introduce some terminology.

A task is considered ready if all of its prerequisites are finished with execution. This is the time when the task can start executing, assuming that its executing node is connected to the server and that that executing node is not currently executing some other task. If the task has no prerequisites, it is considered to be ready at the beginning of the evaluation (time equals to 0).

A prepared task is a task that has all of its prerequisites scheduled. In contrast to unprepared tasks, if all the prerequisites start time are known, then all their finish times are known. In other words, the minimal start time of prepared tasks is the latest finish time of its prerequisites. Assuming, of course, that the information about the completion of the prerequisites has reached the root scheduling server.

The executing node is considered to be ready at the moment when it is connected to the scheduling server and it is not executing any task. This is the moment when that scheduling server can choose to execute some task on it or choose to disconnect it in order to open a connection to some other executing node.

The executing node is considered to be active if it is executing some task. At this point, the scheduling server has to be connected to it and cannot use this connection to do anything else.

The term *free connection* is used when the scheduling server did not open as many connections as it could and there is still more unused memory to open new connections.

The executing node's connect and disconnect times are durations, more specifically, the amount of time that the scheduling server needs to open or close a connection to some executing node. When the scheduling server is connecting or disconnecting from a node, it cannot execute tasks on that particular node or use the memory needed to keep this connection for something else.

A task execution rank is a number that represents task's position in the execution tree Image 3-1. The execution tree is a tree build in such a way, that in the first rank of the tree, there are tasks with no prerequisite tasks. The second rank is constructed of tasks that have a prerequisite only in rank 1. Rank 3 tasks are tasks that have prerequisites only from rank 1 or 2. Other tree ranks are constructed in a similar way.

A node standby time shows how long it will take for a specific executing node to be ready. For the connected nodes, it means the time that it will take to finish executing the current or scheduled tasks. If it is not executing any task, the time is zero. However, if the executing node is not connected to the scheduling server, then this

time is measured to the moment when it becomes connected to the scheduling server. If there are free connections, the time equals to that particular execution node's connection time. Otherwise, the time is a sum of disconnecting and connecting time.

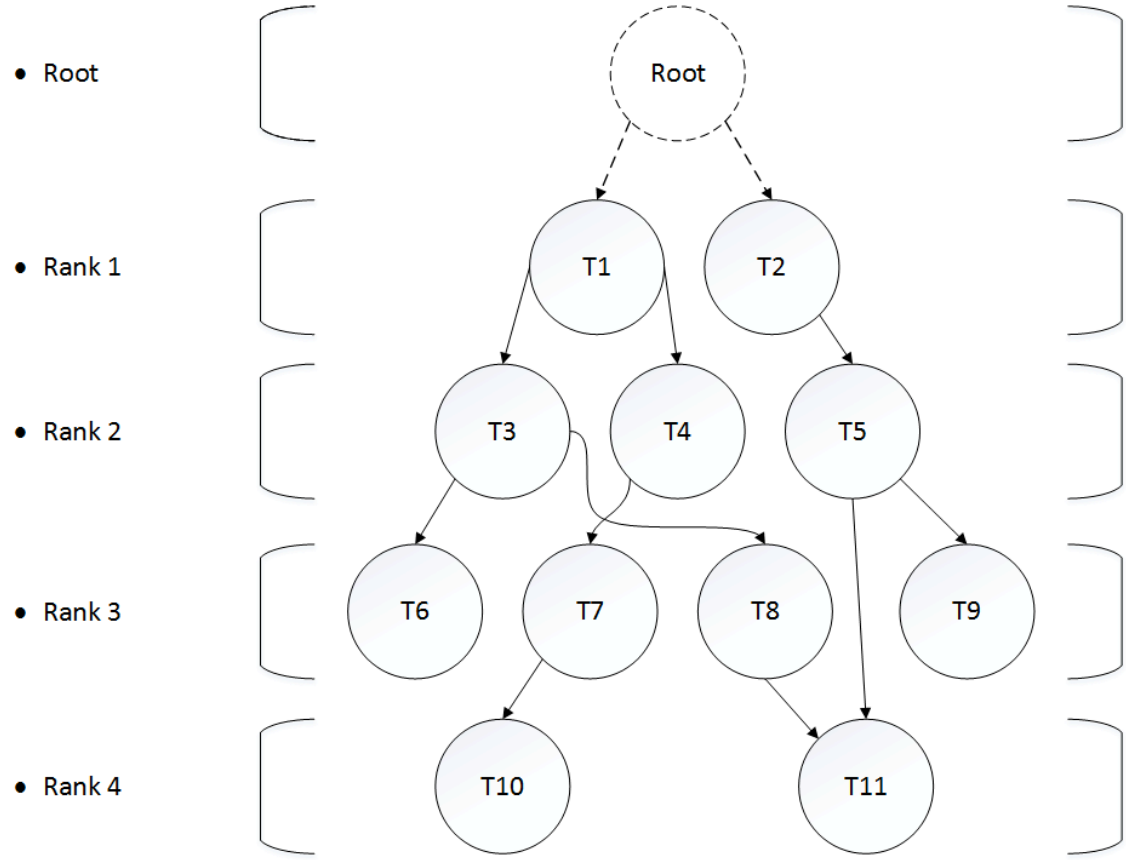

Image 3-1 Task execution ranks

3.1 Hand Written Heuristics

Hand written heuristics were designed before trying to use machine learning techniques to create them. They were created in order to test the evaluation system. They were also used in order to detect more preferable features for machine learning. Since a lot of tests were conducted using hand written heuristics, those heuristics will be described in this section in order to provide a better understanding of how they are used to compare tasks, executing nodes and scheduling servers. Those heuristics are separated into three sections: task comparing heuristics, executing node comparing heuristics and scheduling server comparing heuristics.

3.1.1 Task Scheduling Heuristics

In the first phase of the testing, the entire schedule did not depend on the executing nodes, so a lot of task heuristics were created. The most notable ones are mentioned in Table 3-1. The *max successors* comparison of the tasks depends on the number

of the immediate successors they have. The one with the most successors is the better task. *Earliest ready* finds the task that will become ready first and then schedules it. This one had a problem because many tasks had similar ready time. The next heuristic, *execution tree rank*, was the one that would pick the one with the smallest execution rank (explained in intro). There were also a few others but the best one was comparing multiple things, a composite heuristic if you prefer. The best heuristic checks whether the connection to the executing node is open, that is, if it is for the first task and not for some other, then the first task will be preferred. Next it would try to find the one with the earliest ready time. Finally, if two tasks were equal for the first two conditions, then it would try to find the one with more immediate successors. This heuristic is referred to as the *1. task hand* heuristic.

Table 3-1 Task hand written heuristics

Name	Description
Max successors	Max number of task's immediate successors
Earliest ready	The task that gets ready for the execution first
Exe tree rank	Rank in the execution tree hierarchy
1. task hand	Composite heuristics: is node connected to server, earliest ready time, max successors

3.1.2 Executing Node Heuristics

The second phase was all about comparing the executing nodes. There are more things that can be compared in the executing nodes than in tasks. The most notable heuristics can be found in Table 3-2. The heuristic that was mostly used here (referred to as the 2. F heuristic or just F heuristic) is actually a combination of two things. First, it compares the amount of prepared tasks that the executing nodes have. If the number of the prepared tasks is the same, then it selects the one with the most unfinished tasks. The second heuristic, referred to as the G heuristics, would select the executing node that has the most tasks that are ready when the executing node becomes ready. If those numbers are equal, then this heuristic falls back to 2. F heuristic. The third executing node heuristic, referred to as the 2. H heuristic, concerns itself with successor tasks. To be more specific, it finds its prepared tasks, calculates the number of successor tasks for each one and then it sums all of them. You can say that it is the sum of prepared successor tasks. The node with more prepared successor tasks is assumed to be better. If two executing nodes have equal sums, then this heuristic also falls back to 2. F heuristic.

Table 3-2 Executing node hand written heuristics

Name	Description
F heuristic	Composit: max prepared tasks, max unfinished tasks
G heuristic	Composit: max ready tasks when the node is ready, F heuristics
H heuristic	Composit: sum of the nodes prepared successor tasks, F heuristics

3.1.3 Scheduling Server Heuristics

In the third phase, the main goal was to distribute the tasks among the scheduling servers. The tasks were grouped by their executing nodes so the scheduling servers can decide only about assigning those nodes amongst each other. Here the heuristic needs only to compare the parent scheduling server with its children subtrees. Since there were a lot less targets to pick from, more emphasis was put on the machine learned heuristics and then on the hand written one. There is only one hand written heuristic for assigning the executing nodes to the scheduling servers and it is a pretty simple one, since its function is to test whether the evaluation system is working. It can be found in Table 3-3. This heuristic, referred to as the *3. scheduling servers hand* heuristic, only compares the number of the already assigned executing servers. Every parent scheduling server in the binary tree knows the number of the executing nodes that it assigned to itself and the number of nodes that it assigned to its children. This heuristic would then simply assign a new executing node to the one of those trees with the least assigned nodes. This heuristic is not that effective since it treats its subtrees as single children nodes, which increases the number of the assigned executing nodes to the servers and reduces the number of the assigned nodes to the leaves. The only positive thing about this heuristic is that, by doing this kind of assignment, it reduces the number of sent tasks and task finish information through the binary tree.

Table 3-3 Scheduling serer hand written heuristic

Name	Description
1 scheduling server hand	Num of already assigned executing servers

3.2 Genetic Programming

In artificial intelligence, the genetic programming (GP) is the evolutionary metaheuristic which is inspired by the biological evolution. It is used to find computer

programs (algorithms) that can execute specific tasks. Essentially, GP is a set of instructions and a fitness function. The fitness function is used to determine how well or badly the program performed on a given task. It is a kind of genetic algorithm where every unit has a genotype in form of a tree and machine learning technique which is used to optimize tree population on fitness function which is determined by the ability to perform on the given task. [4]

Genetic programming is considered to be a subtype of the genetic algorithm. Therefore, the genetic algorithm can be steady state and generational. In practice, the steady state is used more frequently because it is often more superior to the generational one and it is easier to implement. In this thesis, the steady state type was used. The initial population is constructed by using a *ramped-half-and-half* method. More about the *ramped-half-and-half* method can be found in [5]. During a crossover, if the creation of a child fails due to the constraints in max nodes or max tree rank, one of its parents is copied with a small plagiarism penalty for the child (1%). Image 1 from [6] is a flowchart of the genetic programming.

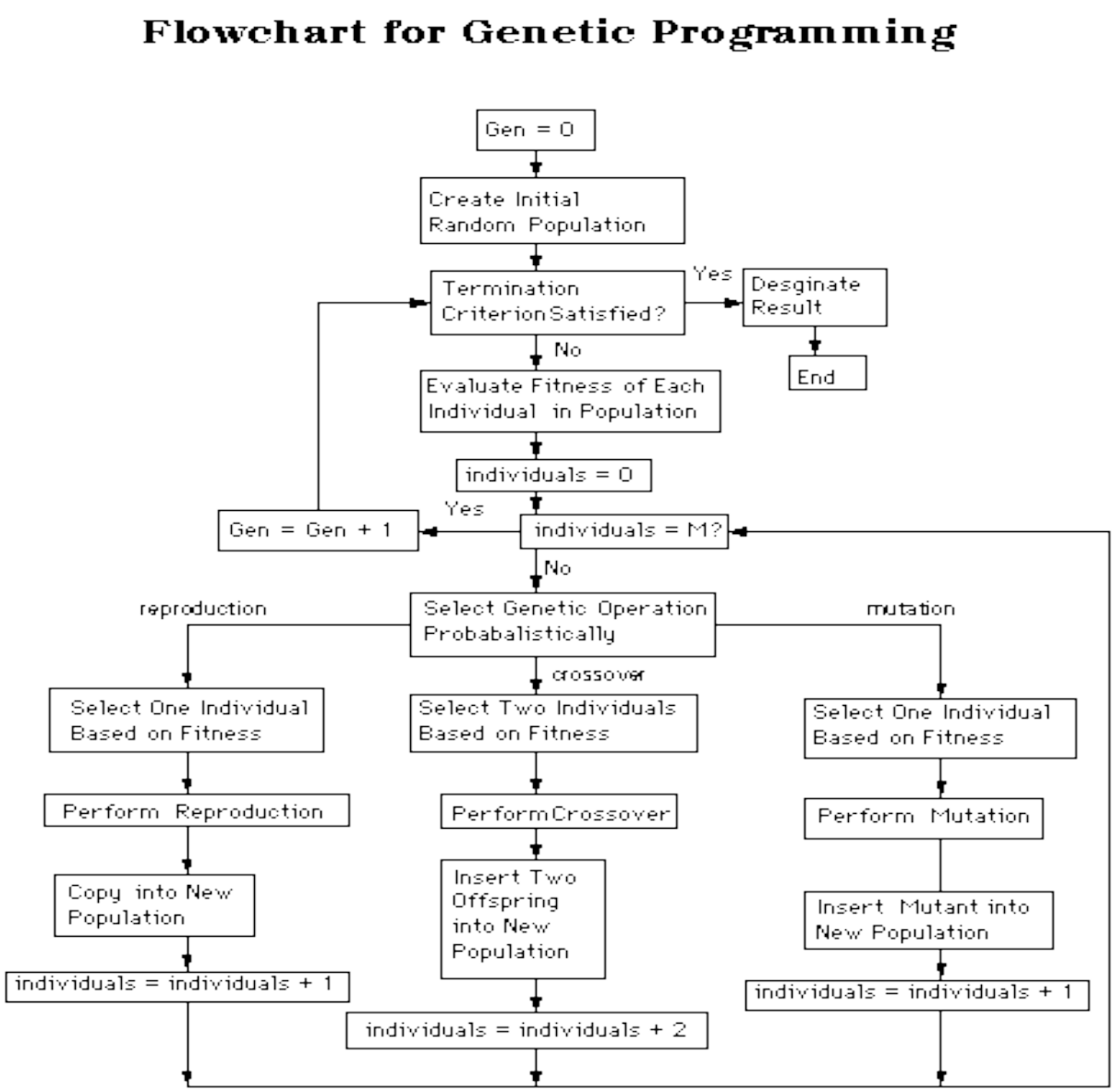

Image 3-2 Flowchart for genetic programming [6]

The tree representation in this thesis is a mix of a symbolic regression and a decision tree.

There are 2 types of input variables: number values array and boolean values array. Output is a single double value which represents a random key encoding of sought heuristic. More about random key encoding can be found in [7].

Function nodes can be unary functions like sin, exp and binary functions like +, -, *, /. Since this is partly a decision tree, there are also if-else nodes which are kind of binary functions that take one random binary input value and depending on whether it is true or false, either the left or the right part of the tree is executed and the other part is discarded.

Leaf nodes can be random constants or values from the input number variables, in which case the node knows the index of the associated input value.

Pseudocode of the used genetic programming algorithm can be seen in Algorithm 3-1.

```
Population = random initial population
Evaluate(Population)
Best = getBest(Population)
While not (stopping criterion)
        Parent1 = tournament(Population)
        Parent2 = tournament(Population)
        Child = crossover(Parent1, Parent2)
        Child = mutation(Child)
        Evaluate(Child)
        Replaced = mockTournament(Population)
        RemoveFromPopulation(Replaced)
        InsertInPopulation(Child)
        If(isBetter(Child,Best)) Best = Child
return Best
```

Algorithm 3-1 Genetic programming

3.3 Cooperative Coevolution

A coevolutionary algorithm is an evolutionary algorithm (or a collection of evolutionary algorithms) in which the fitness of an individual is subjective. It means that the individuals are evaluated based on their interactions with others. Based on the interaction between individuals, coevolutionary algorithms can be divided into two groups: Competitive Coevolutionary Algorithms and Cooperative Coevolutionary Algorithms. [8]

Competitive coevolution is an algorithm where individuals from one population work against individuals from other populations. Those populations can have a same phenotype, for example populations trying to find the best chess player. In that case, each population is a population of chess players and the best individual from the population is the one who plays best against the best players from other populations. Populations can also have different phenotypes, for example a cat and a mouse; the best cat from the population of cats is the one which can catch most mice, and the best mouse is the one who can run away from most cats.

Cooperative Coevolution (CC) is an evolutionary computation method that divides a larger problem into subproblems and solves them independently in order to solve the large problem. [9]

Every one of those subproblems is being solved individually by its own population. Unlike competitive coevolution, here those populations are working together toward a common goal. Usually those populations are evaluated separately and the total best

score for the initial problem can be determined by taking the best individual from each population and combining them.

In this thesis, Cooperative Coevolution is a way to divide a bigger problem into two subproblems. A population was created for each of those two problems, where the individuals from one population represent heuristic for tasks and the individuals from the other population represent heuristic for the executing nodes. The main difference is that, unlike the usual Cooperative Coevolution approach, here those two populations cannot be evaluated separately. During the evaluation, a heuristic for tasks needs to be present as well as a heuristic for executing nodes.

Pseudocode of the implemented Cooperative Coevolution can be seen in Algorithm 3-2.

```
initialise a subpopulation PopTask(S)
initialise a subpopulation PopNode(S)
for i in 1 ..< S {
  evaluate PopTask[i], PopNode[i]
}
find BestTask and BestNode
while termination criteria not satisfied {
  ChildTask = create child from PopTask
  ChildNode = create child from PopNode
  insert / repecate chlidren in populations
  update BestTask and BestNode if needed
}
return BestTask and BestNode
```

Algorithm 3-2 Cooperative Coevolution

3.4 Artificial Neural Networks

In computer science, artificial neural networks (ANN) are computational models inspired by the brain's central nervous system. These models are being used in machine learning and pattern recognition. [11]

Artificial neural networks are representatives of the connectivistic approach to artificial intelligence. The connectivistic approach is based on using lots of simple processing elements. Those process elements by themselves do not show any intelligent features, but when they are assembled together in large numbers, we get a system that presents very interesting features. [12]

In this thesis only the basic form of artificial neural network was used (Image 3-3 from [13]) with sigmoid function [14] as an activation function [13].

More about artificial neural networks can be found in [11] and [12].

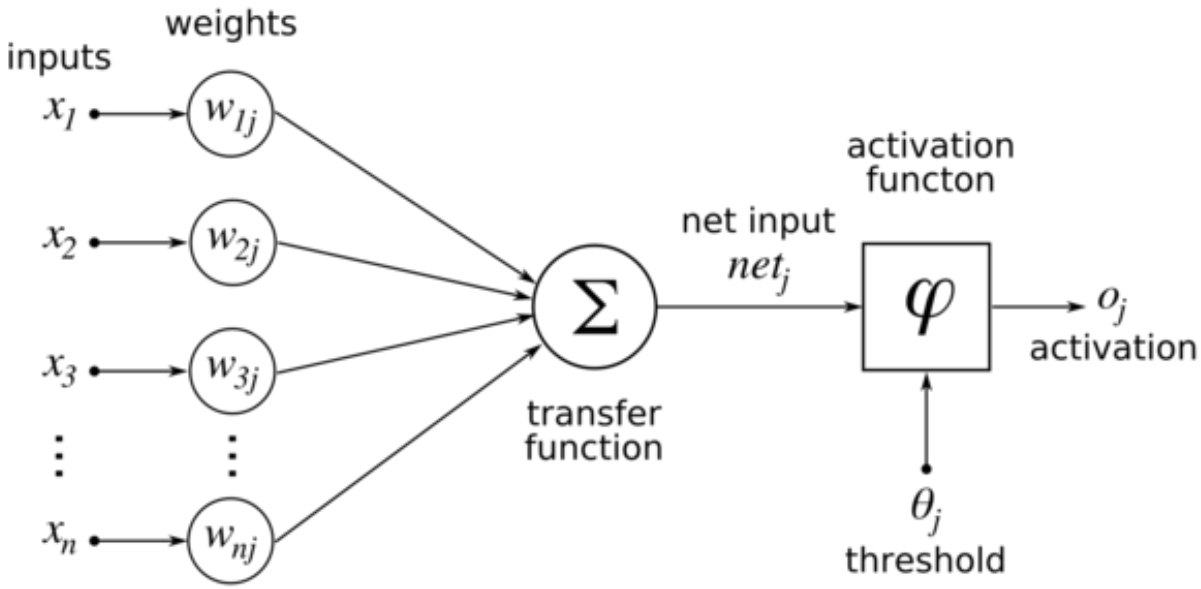

Image 3-3 Artificial neural network [13]

Here the artificial neural network is used for assigning the executing nodes to the scheduling servers. The scheduling servers are stacked as a binary tree, where each server can receive tasks through its parents, so that the root server has all tasks at the beginning of the evaluation. The executing nodes are assigned to the scheduling servers by the tasks that are given to them and the tasks are grouped by the executing nodes. The main decision for a parent here is to decide whether it wants to take a specific executing node for itself or give it to its left or right subtree. Inputs in this artificial neural network represent states of a parent and its children, while outputs are three double values between 0 and 1. The first one belongs to a parent and the other two to its children. The lower the value of the output, the more priority it is given to the scheduling server (parent or children) to get that executing node assigned to it. And of course, scheduling servers that are leaf nodes in binary tree cannot assign the given executing nodes to other servers.

To evaluate a single artificial neural network's weight, a whole evaluation is required. In that single evaluation, the same weights are used every time the evaluation needs to decide to whom to assign a specific executing node. The evaluation result is then assigned as a fitness to those weights. Since we are not comparing outputs from the artificial neural network to target y but having a single performance-like feature, the usual algorithms that operate on a single set of weights like Backpropagation algorithm [15] cannot be used. This thesis used the evolutionary algorithm, specifically Particle swarm optimization (PSO) (the next chapter).

3.5 Particle Swarm Optimization

Particle swarm optimization (PSO) in the computer science is a computational method which optimizes a problem by iteratively trying to improve the individuals based on their fitness. Individuals are candidates for a solution. Their fitness is measured based on their ability to perform well on a given problem. They are moved

around in search space according to simple mathematical formulas using its position, velocity, personal best and global best. This is expected to move the swarm toward the best solutions. [16]

Particle swarm optimization is a nature inspired metaheuristic. It uses a population of particles (individuals) which are moving through the n-dimension search space. They are improving their position by using their own and their neighbors' experience. While determining their movement direction, each individual is using its personal best solution and its local best (social factor) to a certain extent. Individuals' movements are greatly determined by those components' influences. So, having a higher personal score factor leads to diversification and having a higher social factor leads to intensification. This is how this algorithm can combine local search with random search. [12]

Every individual works with and updates its current position X, its velocity V, its personal best Pbest, and local / global best Lbest / Gbest.

Image 3-4 from [17] represents how individuals (particles) are moved during the execution on the algorithm.

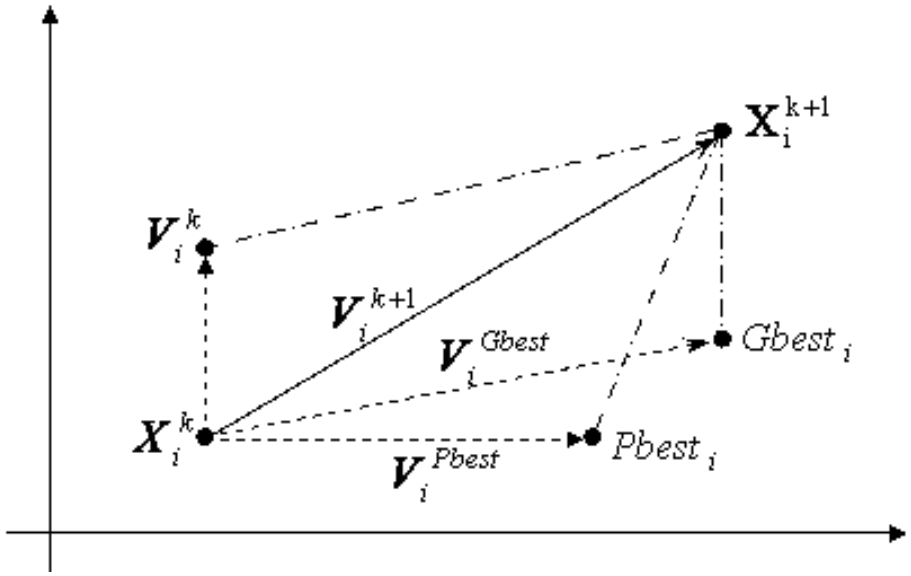

Image 3-4 Movement of individuals [17]

Pseudocode of PSO used in the thesis can be seen in Algorithm 3-3.

```
initialize POP of particles with X and V
set initals X as Pbests
evaluate POP
find Lbest and Gbest

while termination criteria not satisfied {

      for particle in POP {
            particle.V = particle.V + c1 * rand() * Pbest + c2
                        * rand() * LBest
            constraint particle.V to Vmax and Vmin
            particle.X = particle.X + particle.V
      }
      evaluate POP
      update Pbest
      update Lbest and Gbest
}
return Gbest
```

Algorithm 3-3 Particle swarm optimization

In this thesis, PSO is used to train artificial neural networks. Every individual's current position (X) represents one set of ANN's weights, so every individual is potentially one ANN solution.

4. Application of Techniques

All three phases of the testing are covered in this section. In the first phase the scheduling was done on tasks and the executing nodes were regarded just as machines. The system for this phase had only one scheduling server to simplify the initial problem at the beginning. The second phase tackles with the same things as the first one. The difference here is that a new approach had to be taken. The reason for this is that the first phase algorithm was too slow for the real world use. The second phase was also done on a single scheduling server. In the third phase this simplification has been removed and the third phase deals with assigning tasks to the scheduling servers. These scheduling servers are connected in a binary tree.

In each of those phases all their aspects are discussed. Each of those phases had some goals to achieve and there were some problems with the real time application as well. There were more approaches for solving those problems while achieving the main goals of the phase. The most important approaches are explored more in-depth in each phase.

Before exploring all the algorithms created in each phase and their properties, this section talks about how machine learning techniques described in the previous sections are used. They are used to learn heuristics which are actually comparators for tasks, executing nodes and scheduling servers. The heuristics for executing nodes and tasks are more similar than the ones for scheduling servers. All of those heuristics are used in an attempt to optimize (minimize) the makespan. This section also covers the input parameters for the learned models and explains how output parameters are used to schedule tasks.

An example task dataset was created for testing the designed systems in this chapter. The example dataset is very small but efficient for exploring the created algorithms. This chapter will also present results of running the designed systems on this example dataset. Since this dataset is very small, the scheduling server's memory was reduced so whenever something is evaluated on the example dataset, the scheduling server will not be able to have more than two connections open to the executing nodes.

4.1 The Example Dataset

The example dataset consists of 11 tasks which need to be executed on four different executing nodes. The executing nodes are A, B, C and D. The tasks that needed to be executed on the executing node A are 0, 2 and 9. On node B tasks are 4, 5, 8, 11 and on node C tasks are 1, 7 and 10. The tasks needed to be executed on the executing node D are 3 and 6. The execution tree for those tasks can be seen in Image 4-1. Here it means, for example, that task 0 and 3 are prerequisites to task 7, and that task 2 has two successors (task 4 and 5).

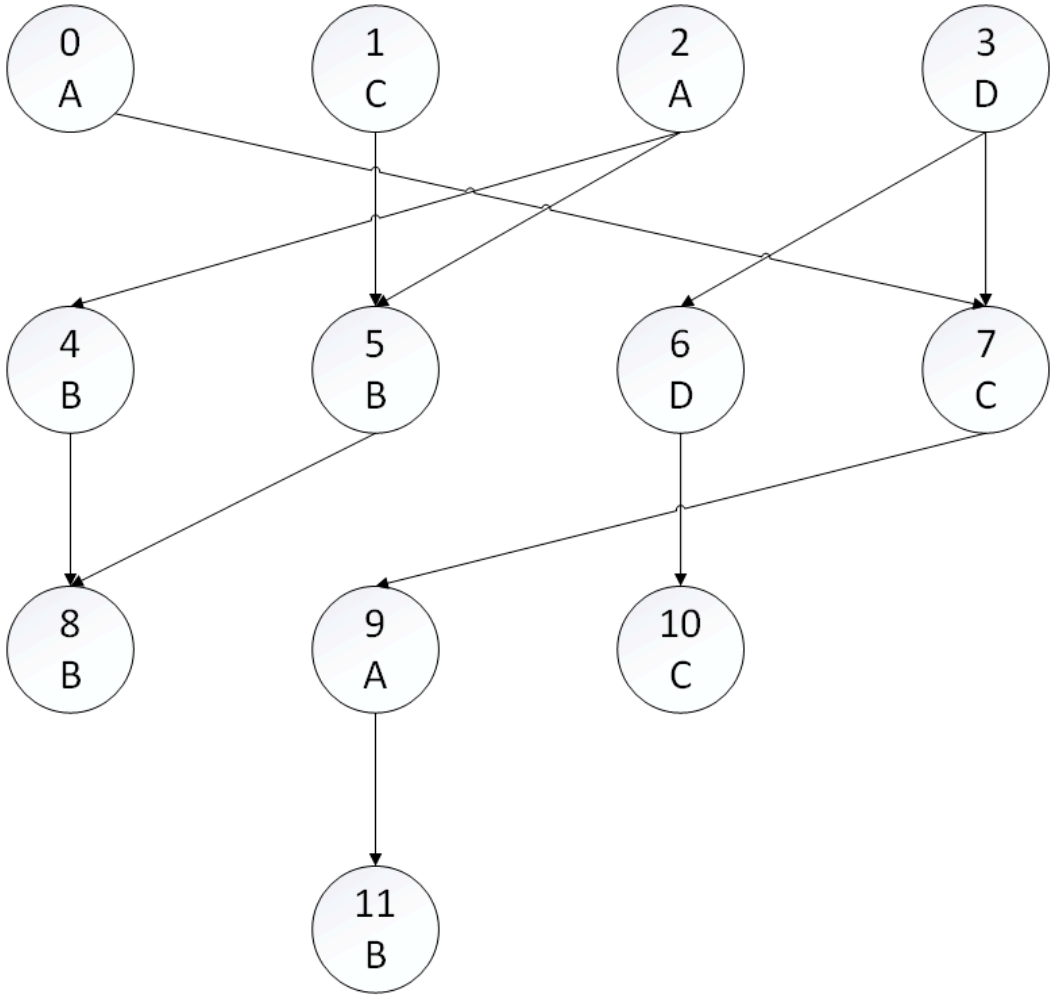

Image 4-1 Example dataset

4.2 Machine Learning Techniques

Machine learning techniques used in this thesis were described in the previous chapter. Since the first phase of the testing was performing very slowly, none of the techniques were used in it. The thing is, to evaluate a single heuristic, one would need to run the entire evaluation system. Since the used metaheuristics are evolution based algorithms, all of them needed to be run a few times in order to get more useful results. And each one of those execution usually means at least few hundred evaluations. So even with faster algorithms like the one in the second phase, where one evaluation would last around 40 minutes with all the input parameters, running evaluation algorithms would take too long. This is the reason why a smaller set of tasks was used for learning heuristics. That smaller set of tasks counts around 10 thousand tasks, which is a lot less than the initial 3 million tasks. For this set of tasks, the evaluation time with the algorithms from the second phase is less than 20 seconds. But if one would try to run this smaller set of tasks with the first section algorithm, it would take around 10 minutes. That is a lot less than 40 minutes but it would still take a few weeks to run a single machine learning technique.

The heuristics that were trying to be learned can be divided into three groups. The first group are the ones that are responsible for selecting the best task out of the list of the tasks assigned to a single executing node. The second group of heuristics are the ones that are used for selecting the best executing node to which the scheduling server will open a connection to. And the third heuristic is used when the scheduling server needs to decide whether it wants to assign a specific executing node to itself or assign it to some of its children subtrees.

Tasks heuristics are created as genetic programs, more exactly as trees that use the task input data and that output a single double value. What types of genetic programming trees were used can be found in the previous chapter. The same tree would be executed for each task and each task would get its key value (the double output value). That key value would then represent inverse priority of a task. The smaller the key value, the higher the task priority is. In other words, the smaller the key value the better the task is. Inputs to the genetic programming tree are number and boolean values. They can be found in Table 4-1. Only one boolean value that is used is: is the task ready. The number input parameters include the number of prerequisites that the task had and the number of its successors. Another input is execution tree rank. The rank was discussed in the previous chapter together with hand created heuristics.

Table 4-1 Inputs for training task heuristics

Name	**Description**
Functions	
Ready	Boolean function, if the task is ready for execution, select left or right subtree
+, -, *, /, sin, cos, exp	Arithmetic operators
Terminals	
Prerequisites	Number, number of task's immediate prerequisites
Successors	Number, number of task's immediate successors
Exe tree rank	Number, rank in execution tree hierarchy

The executing node heuristics are also created as genetic programs. The same tree structures that are used for tasks are used for executing nodes as well. There are real value inputs and boolean inputs. They can be found in Table 4-2. Output is also a single double value that is used in the same way as for tasks; to be the key (inverse priority value) which the scheduling server uses to find the best executing node to open the connection to. There is also only one boolean value used, which is just a coincidence. The boolean value here provides the information whether the executing node is currently connected to the scheduling server. The number of values used as executing node parameters consist of the total number of remaining tasks (that are not scheduled or executed) and the number of prepared tasks. There

is also the number of tasks that are or will be ready before this executing node can disconnect and the number of tasks that are or will be ready before this executing node can disconnect and some other node can connect at its place. The sum of that executing node's prepared tasks' successors is also being used (more about this sum is described in hand heuristics in the previous chapter). The last parameter used here is the executing nodes' local clock (remembers the completion time of the last action performed with the node).

Table 4-2 Inputs for training executing nodes heuristics

Name	**Description**
Functions	
Connected	Boolean function, if the node is connected to the scheduling server, select left or right subtree
+, -, *, /, sin, cos, exp	Arithmetic operators
Terminals	
Connected	Boolean, if the node is connected to the scheduling server
Remaining tasks	Number, total number of remaining tasks
Prepared tasks	Number, total number of currently prepared tasks
Local clock	Number, time of the last action performed with this node
Successors sum	Number, sum of this nodes prepared tasks' successors
Before disconnect tasks	Number, the number of tasks that are or will be ready before this node can disconnect
Before reconnect tasks	Number, the number of tasks that are or will be ready before this node can disconnect and some other node can connect at its place

All the algorithms from the second phase needed both the task and the executing node heuristic to be able to schedule the run of the evaluation. The first tests were done using a fix heuristic for tasks and they tried to learn heuristics for executing nodes. There were also tests that used fixed heuristic for the executing nodes, which tried to learn the heuristic for tasks. But it seemed that the task and the executing node heuristic are not independent. In other words, one task heuristic that is better than the other task heuristic while using one executing node heuristic is not necessarily better than the other task heuristic while using some other executing node heuristic. Cooperative coevolution was used as a response to this. More about the cooperative coevolution was said in the previous chapter. Here CC was used to learn both the task as well as the executing node heuristic at the same time. The idea behind is to let the algorithm try to exploit the dependencies in the task and the executing node heuristic in order to optimize the result.

The third heuristic is somewhat different from the first two. This heuristic is responsible for assigning the executing nodes to the scheduling servers. In the previous heuristics there was a lot of tasks and executing nodes to pick from and the number of the executing nodes and tasks could change dynamically (for example, the executed tasks could be removed). Therefore, in the previous heuristics it would be very hard to make heuristic inputs from all tasks or all executing nodes parameters combined. Here the heuristic has only three scheduling servers to pick from, the parent and its children since the scheduling servers are set up as a binary tree. Now the inputs can be fixed and the heuristic can pick which scheduling server the current executing node can be assigned to. The third heuristic was designed as an artificial neural network which was trained by using the particle swarm optimization algorithm. More about the neural network and the particle swarm optimization was said in the previous chapter. The inputs for this neural network are combinations of the parent and its children parameters. Input parameters for one server can be found in Table 4-3. There are three output nodes, each returning a value of 0 to 1. One for the parent and two for its children. The smaller the output value is, the higher the priority of that scheduling server is to get the executing node assigned to. Each scheduling server's properties consist of the number of the child scheduling servers that it has (it can be 0, 1 or 2). The number of the assigned executing nodes and similarities between the executing nodes that is being assigned with executing nodes that are already assigned to that scheduling server. The number of the assigned executing nodes for the parent refers to the exact amount of the executing nodes it has assigned. For the children it represents all the executing nodes that are assigned to the subtree that starts with that child, since the parent can only see its children and not the entire subtrees. This also applies to the calculating similarities between the executing nodes. One executing node is similar to another if its tasks have a lot of prerequisite tasks in that other executing node. The maximum similarity value from one node to another is 1 and the minimum is 0. Similarities are calculated by using all the tasks at the preprocessing before the evaluations. To calculate the similarities of the executing node and the scheduling server, the similarities from all of the already assigned nodes to a new node are taken and averaged.

Table 4-3 Inputs for training scheduling server heuristics

Name	**Description**
Children	Number, number of child servers in tree
Assigned nodes	Number, number of already assigned executing nodes
Similarities	Number, similarities between executing node that is being assigned and executing nodes that are already assigned to this server

4.3 The First Phase System

Here the idea was to create a system that can effectively schedule tasks on the executing nodes. Some assumptions and some simplifications were made in this phase. The most important simplification here is that multiple scheduling servers were not considered, instead everything was tested as if there was only one scheduling server. The main assumption is that there will be at least one ready task at the beginning of the evaluation and that all the tasks can be executed eventually. This is safe to assume since without it, the makespan would be infinite.

Other characteristic of the designed algorithm is that the scheduling server can simultaneously open connections, close connections and execute tasks on the executing nodes. So when the scheduling server is connecting or disconnecting some executing node, it does not affect other executing nodes. They could be executing tasks, connecting or disconnecting from the scheduling server in a parallel manner.

Since everything is being executed on a single scheduling server, there is no need to send tasks to other servers or to receive messages from other servers. Therefore, the only time that is spent is spent on opening the connections to the executing nodes, closing those connections and executing tasks. Alpha, beta and gamma parameters giving these durations can be found in the main problem description (the second chapter).

In the first phase there were no heuristics for the executing nodes, since the connections to them only depended on the scheduled tasks, but there was a requirement for closing the connections and that did not depend on the scheduled tasks. To simplify things, some well-known methods were used to determine which executing node would be closed. The first method is first in first out (FIFO), which means that when connections had to be closed, the earliest open connection would be closed. The second method is the least recently used (LRU). This method measures how often the tasks are being scheduled to the executing nodes and closes the one to which the tasks were assigned the least recently. The third method closed the least frequently used executing node (LFU). This method knows the time that the tasks were scheduled to the executing nodes and knows which connected executing node will be the one that will be finished first. This method closes the connection to that executing node in order to hopefully minimize the makespan.

Pseudocode of algorithm used in this phase can be seen in Algorithm 4-1.

```
while (has tasks) {

        select prepared tasks from all remaining tasks
        select best task from prepared tasks
        find executing node for best tasks

        if (executing node is connected to scheduling server) {
                schedule best task on executing node
                continue
        }

        if (there are free connections available on scheduling server) {
                start connecting executing node to scheduling server
                schedule best task on executing node
                continue
        }

        find executing node to disconnect
        schedule disconnection of that executing node
        schedule connection of best task's executing node after
                this node had been disconnected
        schedule best task on its executing node
}

time = moment when last task was finished with execution

return time
```

Algorithm 4-1 First phase, Algorithm 1

4.4 The First Phase Performance on Example Dataset

This section presents how the methods applied in the first phase evaluate against the example dataset. Since there are only few tasks in the example dataset, some of the results are similar. All the heuristics used in the first phase are presented in Table 4-4. Those heuristics are described in the third chapter. The methods for determining which executing node to disconnect in order to open the connection to a new executing node are combined with those heuristics.

Table 4-4 The first phase approaches on the example dataset

Phase 1	FIFO	LRU	LFU
Max successors	6100	6050	6050
Earliest ready	4950	6150	4900
Executing tree rank	4950	6150	4900
1. task hand	3750	3750	3750

It is obvious from this table that *1. task hand* heuristic is usually superior to the other heuristics regardless of the methods used for selecting the executing nodes to close. It is also easy to notice that the *earliest ready task* and the *smallest executing tree rank* heuristics are somewhat similar. For this small dataset they provide the same schedules.

Next shown are some images showing Gantt charts explaining how this single scheduling server schedules the tasks from the example dataset using the approaches from the first phase.

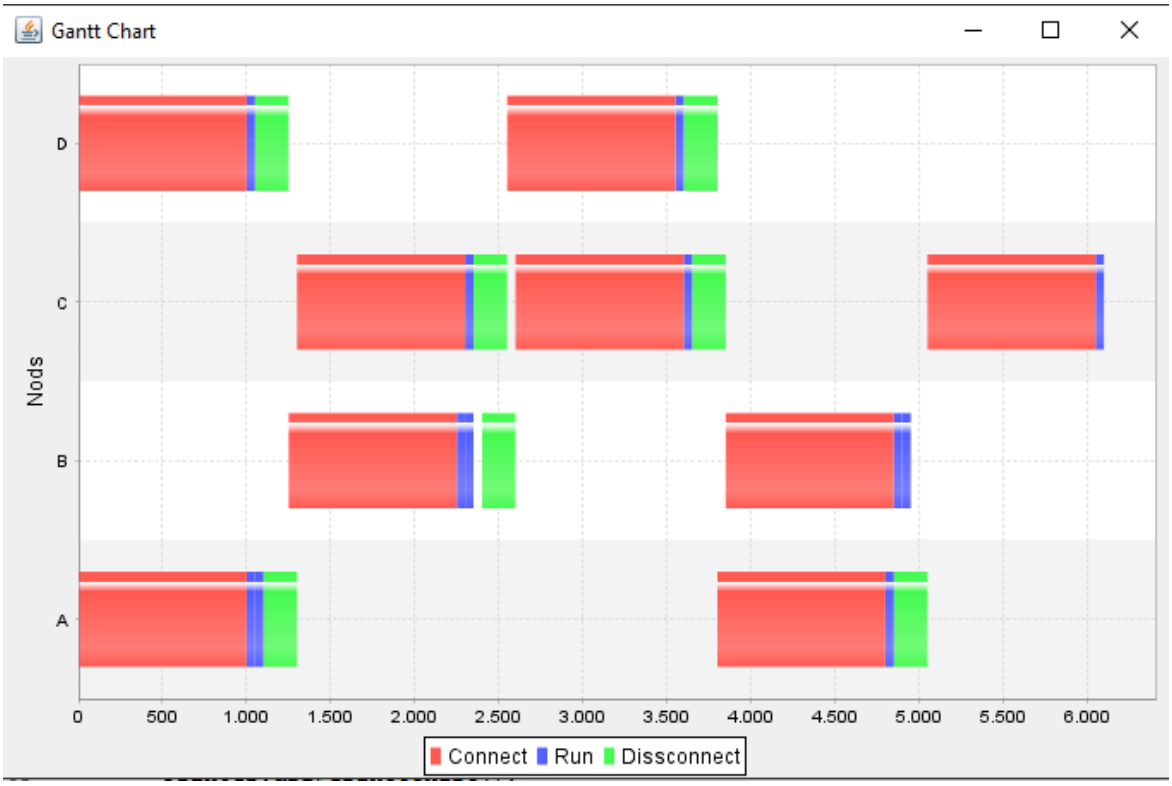

Image 4-2 Max successors heuristic, FIFO method

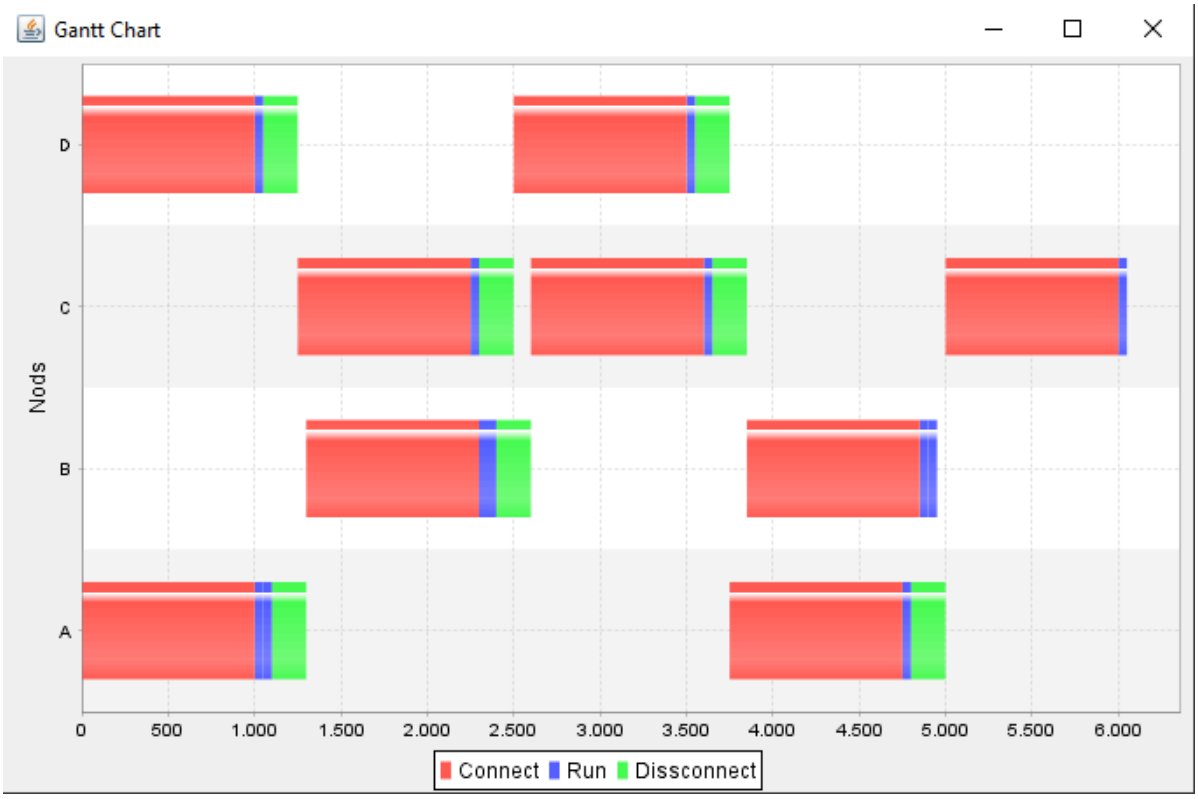

Image 4-3 Max successors heuristic, LRU and LFU methods

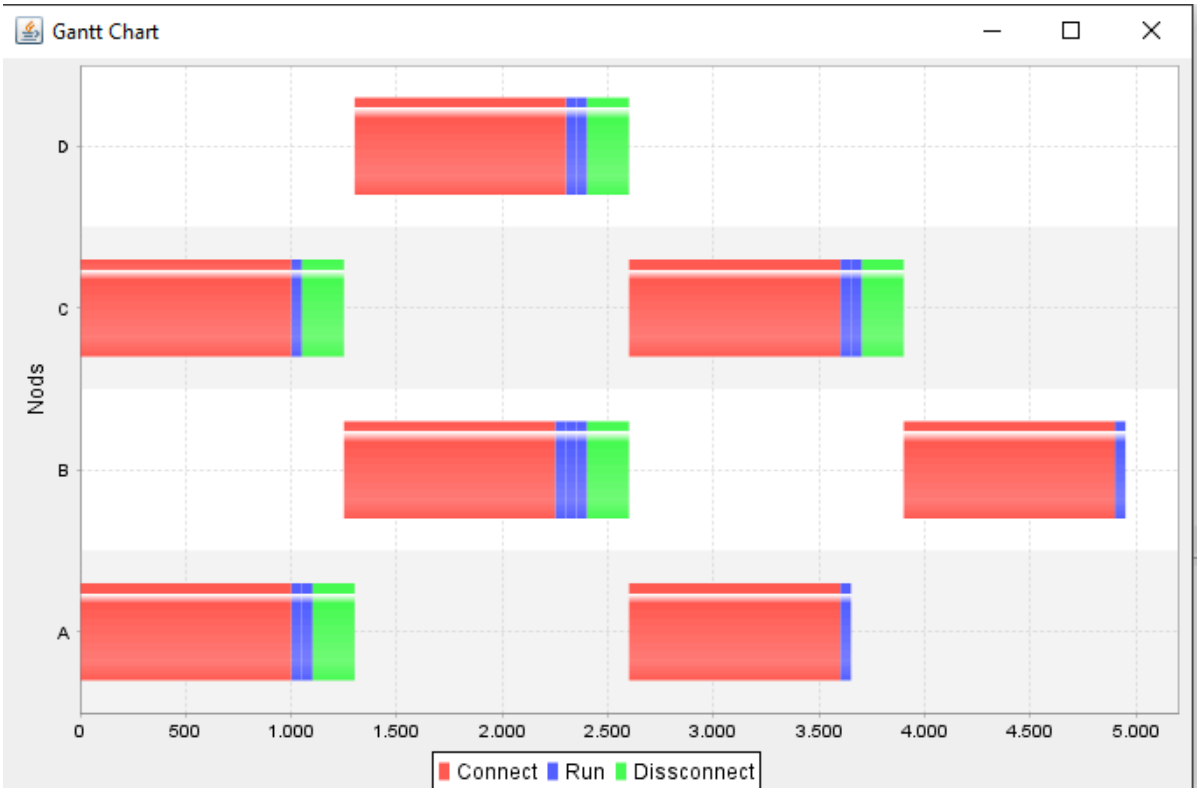

Image 4-4 First ready task and executing tree rank heuristic, FIFO method

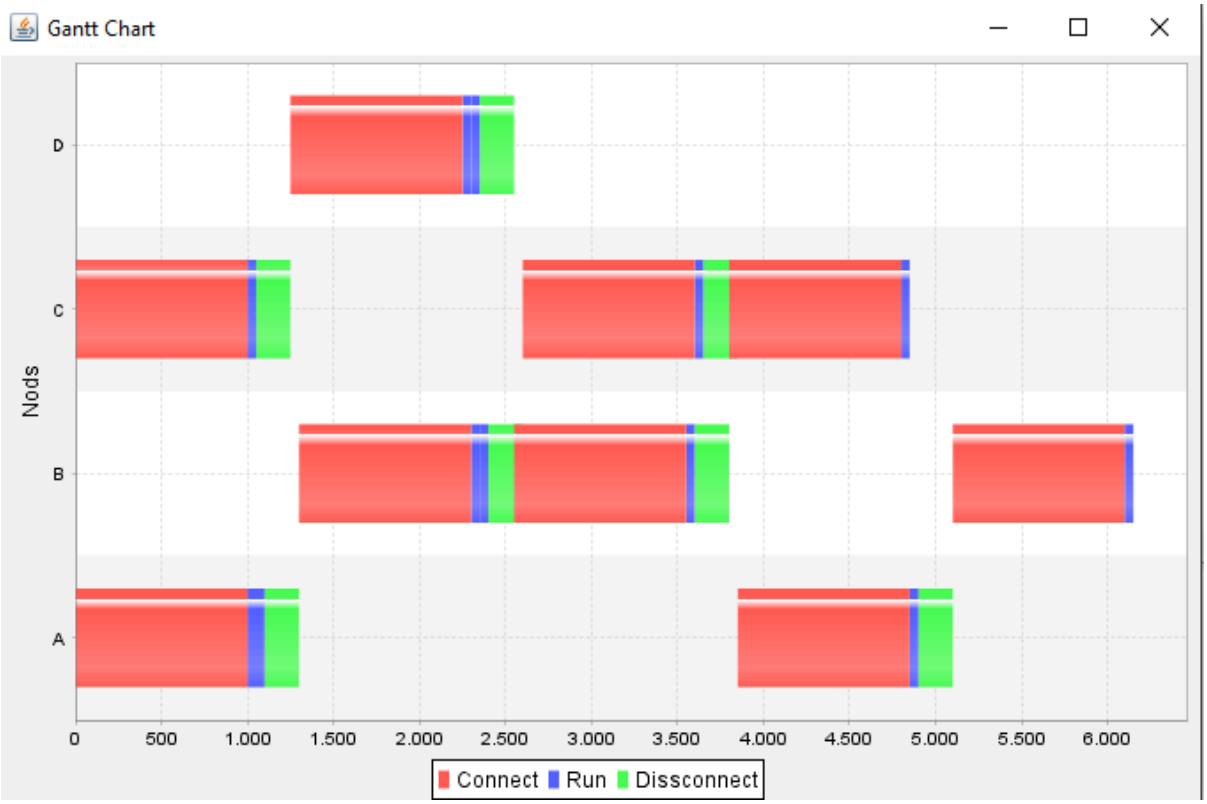

Image 4-5 First ready task and executing tree rank heuristic, LRU method

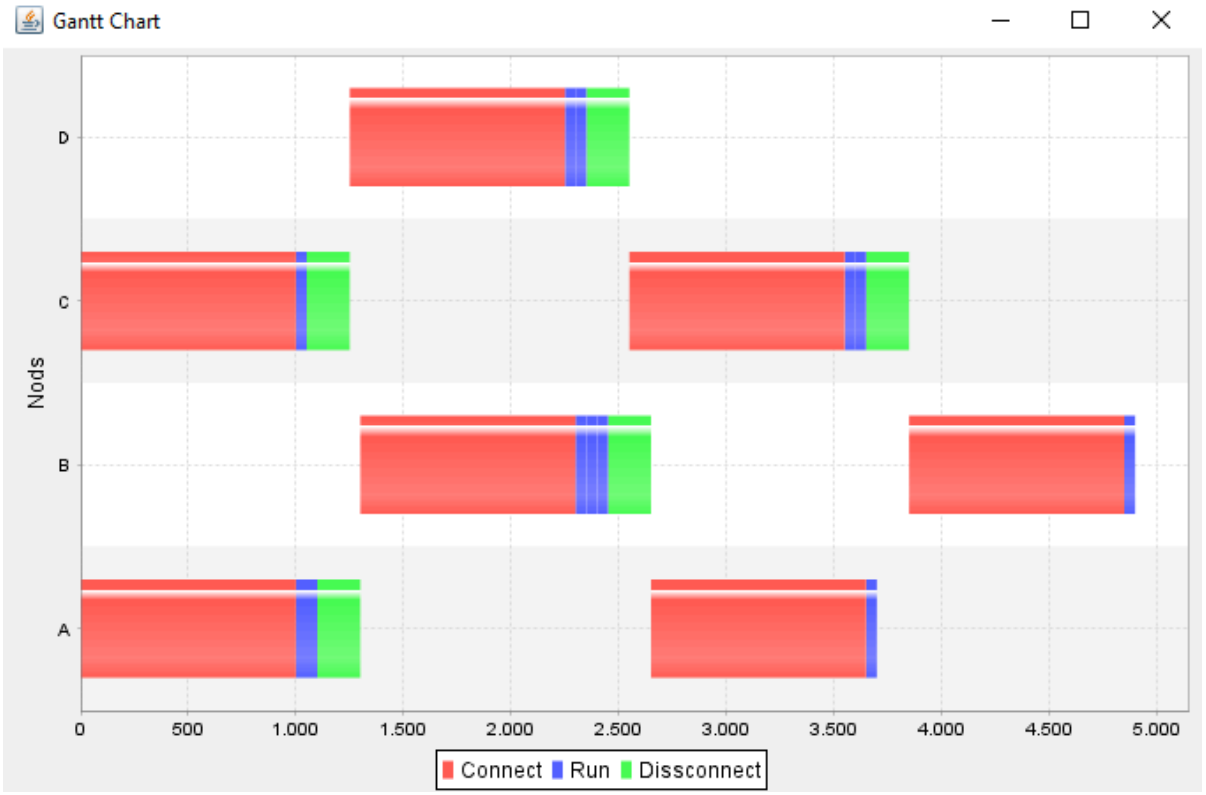

Image 4-6 First ready task and executing tree rank heuristic, LFU method

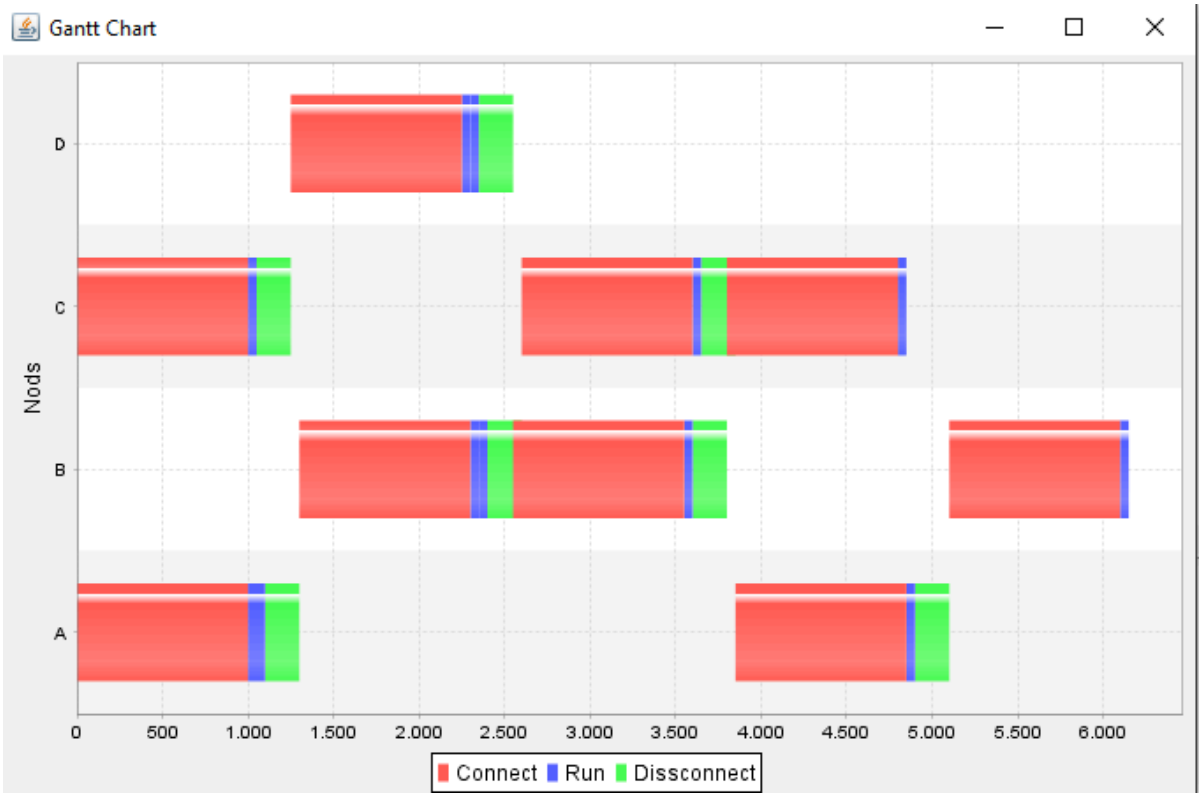

Image 4-7 1. Task hand heuristic, FIFO, LRU and LFU methods

4.5 The Second Phase System

Since the first phase had a big problem of not being able to evaluate all the tasks in a realistic setting in less than an hour, there was a need for a faster evaluation. A new evaluation algorithm needed to be devised. The main problem includes a lot more tasks (around 3 million) than executing servers (around 30 thousand). The idea for the second phase approaches comes from reversing the priority order. The first one would need to find the best executing node, open the connection to it and then proceed in trying to find the best task for this executing node.

As in the first phase, we are still using only one scheduling server, so again there is no need for sending the tasks or the task finish information through the scheduling server tree. Also, that scheduling server should be able to open and close the connections to the nodes and execute tasks on them in parallel.

The first attempt to create this system had the most general algorithm which would allow the executing nodes to be switched (disconnected) as soon as they became ready. It used two comparator heuristics. The first one was used for selecting the best executing node and the second one for selecting the best executing task. That executing task was selected from all the tasks that were assigned to the chosen executing node. The idea here was to filter the tasks on the executing nodes and when searching for the best task, you would have to look only through around 100

tasks (3 million tasks in total / 30 thousand executing nodes), which is a lot less than 3 million tasks.

So now instead of searching through 3 million tasks, the algorithm was searching through 30 thousand executing nodes and then through 100 tasks on average in each iteration. Of course, in every iteration we would remove one task, but it still needed to go through a lot of elements. One disadvantage to real work evaluation duration is that the executing nodes had more parameters than the tasks and it takes more computation power to compare two executing nodes than to compare two tasks. This was not going to be modified since those parameters are used to find better executing nodes.

In the first phase there was a need for a method that would be responsible for deciding which executing node should be disconnected from the scheduling server. In this phase there is no need for that method since the executing node heuristic acts as a comparator. To determine which node to disconnect when there is a need to open a new connection to some other executing node, the algorithm can use the provided executing node comparator to find the worst executing node. Of course, first it needs to filter only the executing nodes that are connected to the scheduling server.

The algorithm in the first attempt was faster than the original algorithm from the first phase that was only comparing the tasks. This algorithm was a move towards the right direction but it was still too slow to evaluate all 3 million tasks in less than an hour. Its biggest problem is that in each iteration it was searching through all the executing nodes and then through all the tasks in that executing node in order to find one task that was going to be scheduled.

Pseudocode of the first attempt algorithm (later referred to as algorithm 2) can be seen in Algorithm 4-2.

```
func schedule best task for node {
        select best task for node (heuristic for tasks)
        start / schedule best task at earliest moment when node is ready
                (update node ready time and times for task's successors)
}

main_time = 0;
while(has tasks){
    select set of ready nodes that have ready tasks
    add nodes:
        - nodes that have tasks that are ready before node can disconnect
                AND (are not active OR in closing)

    if (set is empty) {
        main_time = earliest moment when (any task is ready AND (can
                open new connection OR that task's node is also ready))
        continue;
    }

    select best node from set (heuristic for nodes)

    if (selected node is ready) {
        schedule best task for node
        continue;
    }

    if can open new connection {
        start connecting best node
        schedule best task for node after the connection is opened
    }
    else {
        select ready node to disconnect (heuristic for nodes)
        start disconnecting node
        start connection of best node after previous node has been
            disconnected
        schedule best task for node after best node is connected
    }
return moment when last task was finished with execution
```

Algorithm 4-2 Second phase, Algorithm 2

Since the first attempt of creating a faster algorithm was not that successful because it was not able to evaluate a larger number of tasks in the expected time, some modifications had to be made. Since opening and closing the executing nodes is a lot more expensive than executing tasks, the idea was to check for new executing nodes less often. It means that comparing the executing nodes had to be done less often and scheduling more tasks on a node more often. In this approach, once connected executing node will remain connected to the scheduling server as long as there are ready tasks present for that node or tasks that will be ready (their prerequisites will finish executing) before that executing node could be disconnected. This leads to the second attempt. In the second attempt the scheduling server's free connections were filled only with ready executing nodes. If there were not enough ready executing

nodes to be put in the free connections, those connections would not be filled with the next best (provided by heuristic) executing nodes.

Pseudocode of the second attempt algorithm (later referred to as algorithm 3) can be found in Algorithm 4-3.

```
main_time = 0;
while(has tasks) {

    can_perform = select a set of open / ready nodes with any task that
        will be ready before node can disconnect

    for node in can_perform {

        select best task for node (heuristic for tasks)
        run / schedule best task for that node (update node ready time
                and times for task's successors);
    }
    to_close = select a set of other open / ready nodes with no ready
                tasks before node can disconnect

    for node in to_close {
        start disconnecting node
    }

    while (new connection can be opened) {
        can_open = select a set of closed nodes with any task that will
            be ready before node can re-connect
        if (can_open.isEmpty) {
            break
        }
        select best node from can_open
        start connecting best node
        update ready time of best node
    }

    main_time = earliest moment when (any task is ready AND (new
        connection can be opened OR that task's node is also ready))
}

return moment when last task was finished with execution
```

Algorithm 4-3 Second phase, Algorithm 3

This algorithm actually had everything that was needed for this phase. It can evaluate 3 million tasks in less than an hour. It handles parallel connections to the scheduling server and does not need to use any heuristic or sophisticated method to determine which connection to close. There is no need for that since it keeps the unused connections always closed.

But this is not the only way in which such an algorithm can be made. For this thesis, four more scheduling server algorithms have been created in order to see which approach produces better results. Of those four other approaches, three are similar to this algorithm 3 in many ways. The fourth one is somewhat different but it is based on similar approaches to compare the execution nodes before comparing the tasks and to keep the connection to the specific executing node alive until there are no more ready tasks on that node.

The next is algorithm 4 and it looks a lot like algorithm 3. The main difference between algorithm 3 and 4 is that algorithm 4 tries to have no free connections. Therefore, it opens the best nodes based on the heuristic even if they will not have ready tasks when they are connected to the server. It can also prevent the scheduling server to close the executing node when that node finds itself in *to_close* set if the priority of that node is high enough. The reasoning behind this approach lies in the idea that if the heuristic is good enough, then it can somehow try to predict which executing node will be used soon. If it is successful, it can save time by preparing the executing nodes before they are actually needed.

Pseudocode of algorithm 4 can be found in Algorithm 4-4.

```
main_time = 0;
while(has tasks) {

    can_perform = select a set of open / ready nodes with any task that
        will be ready before node can disconnect

    for node in can_perform {

        select best task for node (heuristic for tasks)
        run / schedule best task for that node (update node ready time
                and times for task's successors);
    }
    to_close = select a set of other open / ready nodes with no ready
                tasks before node can disconnect

    for node in to_close {
        start disconnecting node
    }

    while (new connection can be opened) {
        can_open = select a set of closed nodes with any task that will
            be ready before node can re-connect
        if (can_open.isEmpty) {
            break
        }
        select best node from can_open
        start connecting best node
        update ready time of best node
    }

    if (can connect more node) {
        select best node from closed nodes
        start connecting best node
        update ready time of best node
    }

    main_time = earliest moment when (any task is ready AND (new
        connection can be opened OR that task's node is also ready))
}

return moment when last task was finished with execution
```

Algorithm 4-4 Second phase, Algorithm 4

The fifth algorithm functions on similar principles as the previous two but is essentially much different. Similar to the previous approaches, it starts by opening as many connections from the scheduling server to the executing node as it can. The previous approaches would then try to find the best executing node in each iteration from the open nodes and schedule the execution of its best task. However, this approach schedules all ready tasks for that executing node depending on its local clock.

Executing node's local clock represents the finish times of the last actions scheduled on that node. An action can be opening a connection to the node, executing a task

on it and closing a connection to that node. The scheduling server keeps its open nodes in a list sorted on node's local clocks. Since in the previous versions everything was synchronized around the *main_time* which was kind of a global clock, in this version everything is synchronized amongst multiple local clocks. Here there is a list of sorted integers that represents free connections (*sorted_clocks*). When all the connections to the nodes are closed, this list holds as many integers as there can be open connections but when all the connections are open, it is an empty list. So in the beginning of the evaluation this is a list of zeroes. When the scheduling server wants to open a connection to the executing node, it takes the first (earliest) time from the list, sums it with executing node's connection time and sets the node's local clock to that time. While closing a connection to the executing node, the algorithm takes closing node's local time, sums it with disconnect time and inserts it into *sorted_clocks* list. Of course, one executing node cannot start connecting using a connection if it is disconnecting on another.

This approach also has its open executing nodes in the sorted list. This list is sorted on their local clocks. In each iteration, this algorithm takes the first executing node (one with the smallest local clock) and schedules all ready tasks that are assigned to it. Then it updates the node's local clock and inserts it back to the open node list. The only exception from the returning to the open list is when all the tasks for that executing node have been scheduled. In that case we can schedule its disconnection, because when it finishes with the scheduled tasks, it will be useful to automatically replace it with the executing node that still has tasks to execute.

Pseudocode of algorithm 5 can be found in Algorithm 4-5.

```
func substitute_node(node_to_close,
                     andOpenNewNodeFrom: all_executing_nodes,
                     putNewNodeIn: executing_node_list
                     sortedClocks: sorted_clocks) {

  close_connection(node_to_close)
  update sorted_clocks with connection closed time
  new_node = all_executing_nodes.remove_best_node()// heuristic for
             nodes
  schedule_open_connection(new_node) using sorted_clocks.pop()

  executing_node_list.insert_in_list_sorted_on_min_ready_time(new_node)
        // sorted on min ready time
  all_executing_nodes.add(node_to_close)
}

sorted_clocks = SortedIntegerList()
// when it is possible to open connection, max size of
// max_num_of_connections
executing_node_list = best_n_from_comparator(all_executing_nodes,
        max_num_of_connections)
// sorted on min ready time, then on heuristic
all_executing_nodes.remove(executing_node_list)
open all nodes in executing_node_list;

while(task_list.size > 0) {

  executing_node = executing_node_list.pop_first()
  // if node has no prepared tasks
  if (executing_node.num_of_prepared_tasks() <= 0) {
        substitute_node(executing_node,
                        andOpenNewNodeFrom: all_executing_nodes,
                        putNewNodeIn: executing_node_list
                        sortedClocks: sorted_clocks)
        continue
  }

  execute_tasks_for_node(executing_node)
  // heuristic for tasks; scheduling all tasks that are ready;
  // update local clock (node ready time)

  // total number of remaining (unfinished) tasks is equal to 0 happens
  // only near the end
  if (executing_node.num_of_tasks == 0) {
        substitute_node(executing_node,
                        andOpenNewNodeFrom: all_executing_nodes,
                        putNewNodeIn: executing_node_list
                        sortedClocks: sorted_clocks)
        continue
  }

  executing_node_list.
        insert_in_list_sorted_on_min_local_clock(executing_node)
  // sorted on min ready time
}
return moment when last task was finished with execution
```

Algorithm 4-5 Second phase, Algorithm 5

The next algorithm is very similar to the first two. The difference is in *can_perform* executing node set or to be more specific, in its creation. In the third algorithm this set consists of open executing nodes with tasks that are or will be ready before that node can disconnect. In this algorithm, it consists of open nodes with tasks that are or will be ready before the node can disconnect and another node can connect in its place. The reasoning behind this approach is kind of obvious. If we decide to replace the executing node that is connected to the scheduling server, the earliest time that any task could be executed on that connection is when the newly connected node finishes with connecting. So it might be useful to wait for the currently open node to execute as many tasks as it can before disconnecting it.

Pseudocode of algorithm 6 can be found in Algorithm 4-6.

```
main_time = 0;
while(has tasks) {

    can_perform = select a set of open / ready nodes with any task that
        will be ready before re-connect time

    for node in can_perform {

        select best task for node (heuristic for tasks)
        run / schedule best task for that node (update node ready time
                and times for task's successors);
    }
    to_close = select a set of other open / ready nodes with no ready
                tasks before re-connect time

    for node in to_close {
        start disconnecting node
    }

    while (new connection can be opened) {
        can_open = select a set of closed nodes with any task that will
            be ready before node can re-connect
        if (can_open.isEmpty) {
            break
        }
        select best node from can_open
        start connecting best node
        update ready time of best node
    }

    main_time = earliest moment when (any task is ready AND (new
        connection can be opened OR that task's node is also ready))
}

return moment when last task was finished with execution
```

Algorithm 4-6 Second phase, Algorithm 6

The final algorithm that was used is only a combination of modifications that algorithm 4 and 6 did on algorithm 3. So this algorithm 7 tries not to have free connections but as much connected nodes in parallel as possible. It also creates its *can_perform* executing node set as a set that consists of executing nodes that have tasks that are ready or will be ready before this executing node can disconnect and the scheduling server can connect to some other node in its place. Both of those things are used here in an attempt to minimize the makespan by reducing the disconnecting and connecting of the executing tasks to the scheduling servers in favor of waiting for the tasks to become ready and pre-connecting the executing nodes even if they might not be the best choice.

Pseudocode of algorithm 7 can be found in Algorithm 4-7.

```
main_time = 0;
while(has tasks) {

    can_perform = select a set of open / ready nodes with any task that
        will be ready before re-connect time

    for node in can_perform {

        select best task for node (heuristic for tasks)
        run / schedule best task for that node (update node ready time
                and times for task's successors);
    }
    to_close = select a set of other open / ready nodes with no ready
                tasks before re-connect time

    for node in to_close {
        start disconnecting node
    }

    while (new connection can be opened) {
        can_open = select a set of closed nodes with any task that will
            be ready before node can re-connect
        if (can_open.isEmpty) {
            break
        }
        select best node from can_open
        start connecting best node
        update ready time of best node
    }

    if (can connect more node) {
        select best node from closed nodes
        start connecting best node
        update ready time of best node
    }

    main_time = earliest moment when (any task is ready AND (new
        connection can be opened OR that task's node is also ready))
}

return moment when last task was finished with execution
```

Algorithm 4-7 Second phase, Algorithm 7

4.6 The Second Phase Performance on Example Dataset

In this section it is explained how well the algorithms designed in the second phase are performing on the example dataset. Some heuristics created for this phase are also presented here. The designed algorithms are described in the fourth section and hand created heuristics are presented in the third section. In Table 4-5 some machine learned heuristics are also presented.

Table 4-5 The second phase approaches on the example dataset

Phase 2	alg 2	alg 3	alg 4	alg 5	alg 6	alg 7
F node, 1 hand task	4950	4950	6100	6050	4950	6100
G node, 1 hand task	5000	4900	4900	4900	4950	6100
H node, 1 hand task	6200	3750	4900	3750	3750	4900
Cooperative coevolution	---	3750	6100	3750	3750	4900

Each algorithm is one column and one combination of the heuristics is presented in rows in this table. Each row is a combination of one node heuristic and one task heuristic. Only one task heuristic has been used (*1. hand task* heuristic) for all hand designed executing node heuristics. The row that states cooperative coevolution presents a mix of two heuristics (the executing node and task heuristic) that were learned together. Those learned heuristics were not learned using this example dataset, but using a smaller version of the normal dataset (around 10 thousand tasks). In each column, cooperative coevolution represents the heuristics that were learned while evaluating by using the column algorithm. For example, cooperative coevolution and algorithm 3 column represent the executing node and task heuristics that were learned on the smaller version on the normal dataset, while evaluating that dataset using algorithm 3. Algorithm 2 was not used for learning heuristics because it takes too long to execute.

One can see here that the machine learned heuristics are just as good as the best combinations of hand written heuristics for five of the six designed algorithms. The main reason why this is positive is that those heuristics were not learned using this dataset. This can mean that the learned heuristics are able to generalize to some extent.

Now some images of Gantt charts follow showing how this single scheduling server schedules the tasks from the example dataset using the approaches from the second phase.

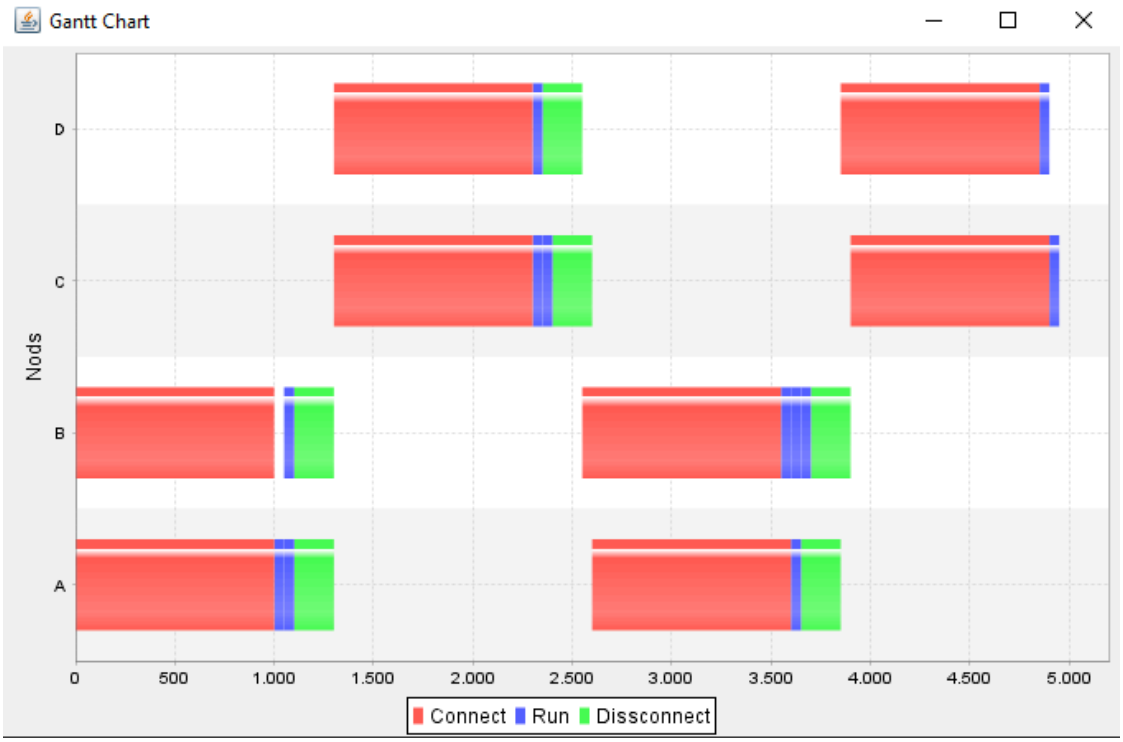

Image 4-8 Algorithm 2, F executing node heuristic, 1. hand task heuristic

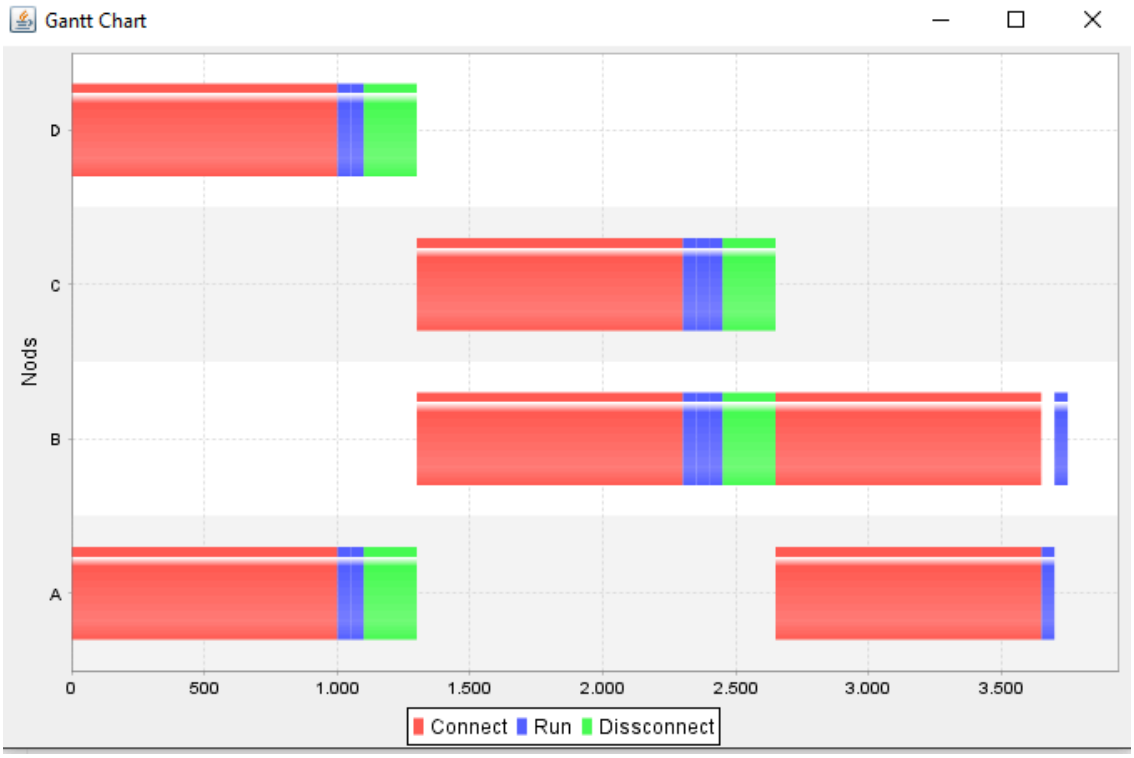

Image 4-9 Algorithm 3, 5 and 6, H node, 1. hand task heuristic and also Algorithm 3, 5 and 6 cooperative coevolution

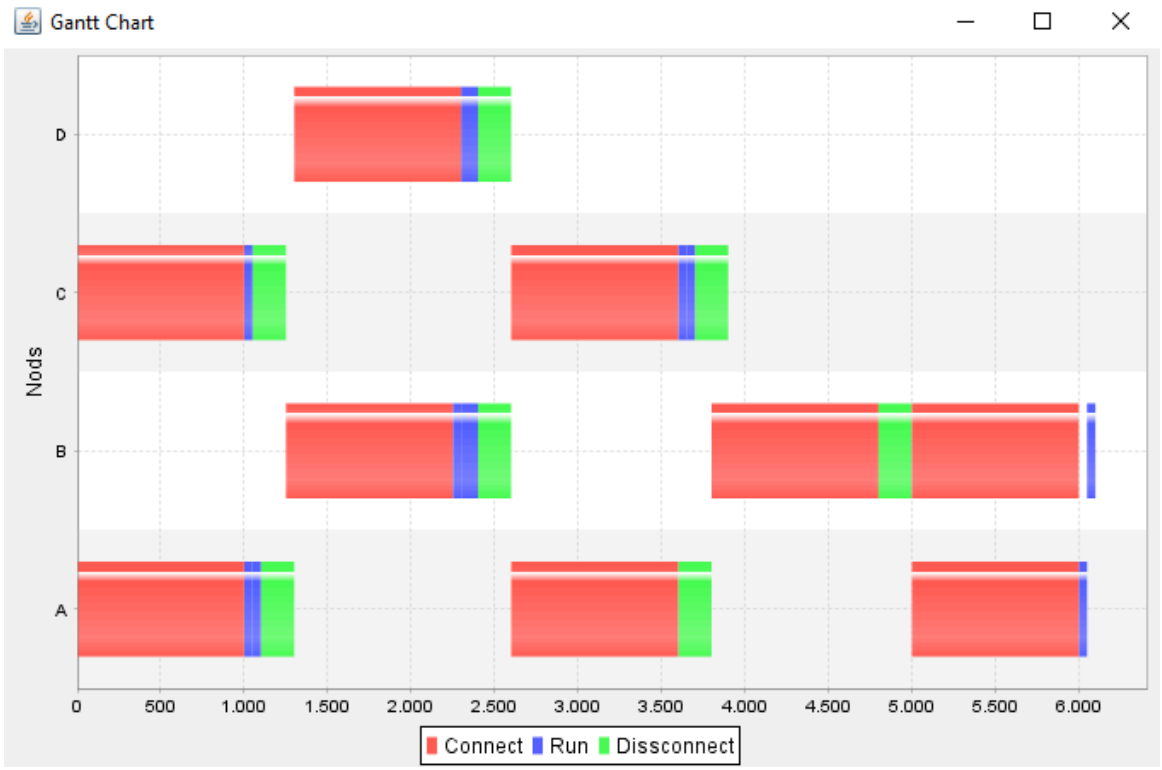

Image 4-10 Algorithm 4, F node, 1. hand task heuristic and also Algorithm 4 cooperative coevolution

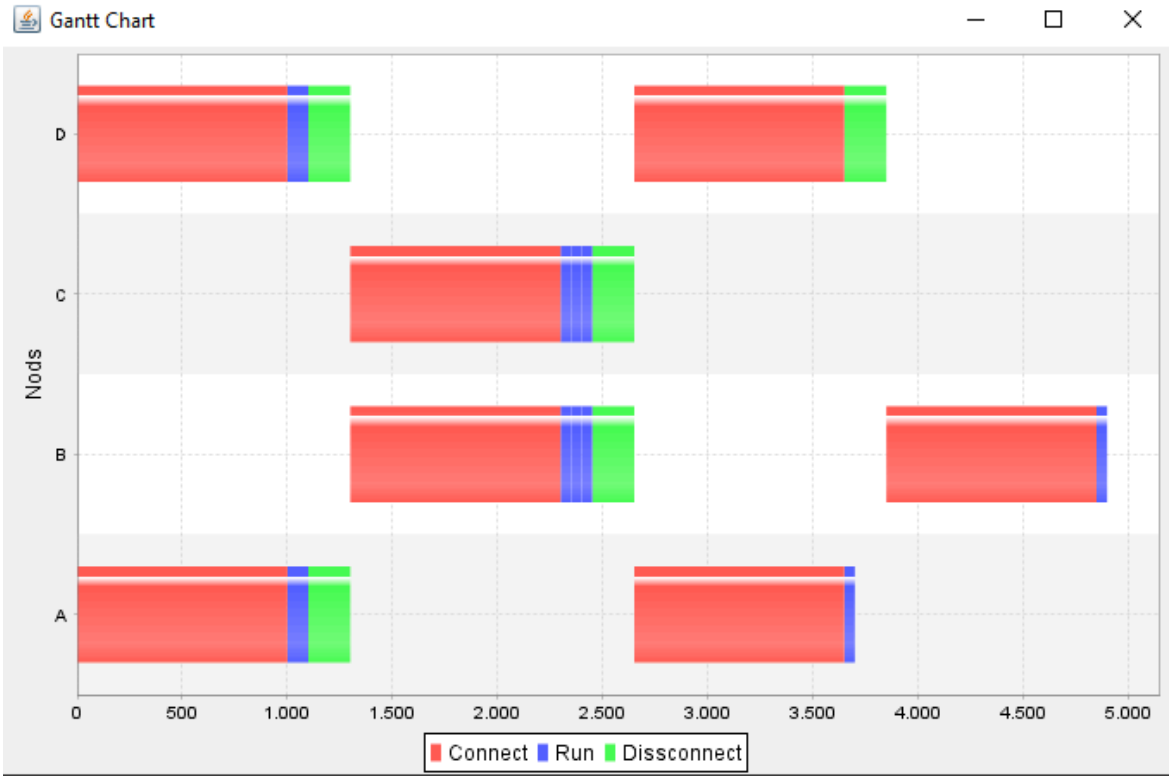

Image 4-11 Algorithm 4, H executing node heuristic, 1. hand task heuristic

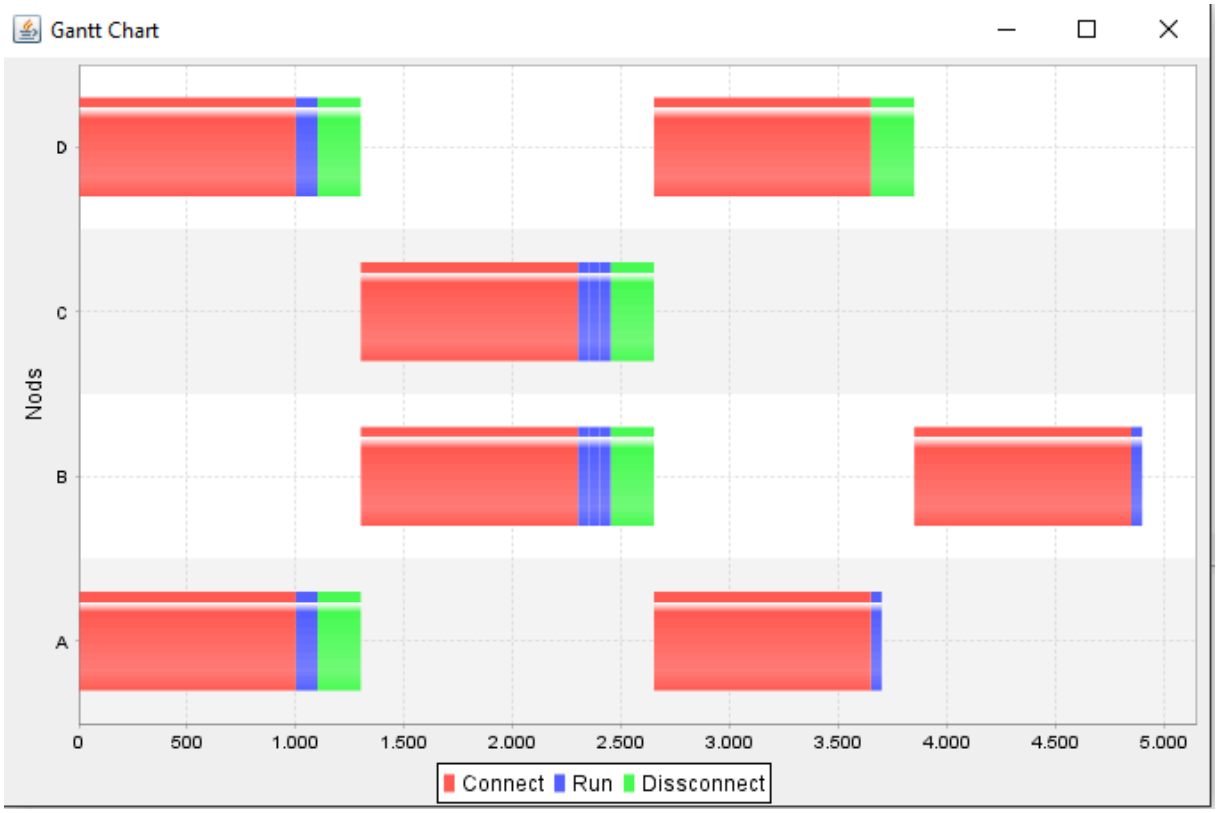

Image 4-12 Algorithm 7, F node, 1. hand task heuristic and also Algorithm 7 cooperative coevolution

4.7 The Third Phase System

The third phase was all about removing the single scheduling server simplification, since everything that was done on a single server was tackled with in the first and the second phase. In this phase, multiple scheduling servers are allowed. They communicate via messages. Those messages can be sending the task to children or sending the information about the task finish time to the parent. When the scheduling server is sending tasks to its children, it groups them in batches in order to send multiple tasks faster, rather than sending a task by task. The time required to send one task one level down the binary tree takes 10 ms (*delta* properties from the second chapter), but when sending multiple tasks in a batch, two tasks can be sent one after another in a time span of 0.2 ms.

In this system, it is important for all the tasks that have prerequisites to wait on the root scheduling server until they become prepared, that is, until all of their prerequisites are scheduled. Only when the tasks are prepared, the root scheduling server can assign those tasks to itself or one of its children scheduling servers. Those children scheduling servers can then assign it to themselves or pass those tasks to their children. Only leaf scheduling servers cannot pass tasks to somebody else, they must assign those tasks to themselves.

Since real time evaluation is required (must be shorter than an hour) the scheduling server groups tasks by their executing nodes. This also helps to schedule the tasks on the executing nodes, since the scheduling server schedules the assigned tasks to the executing nodes using the algorithms from the second phase. Those algorithms then schedule the executing nodes first and then the tasks. So when the scheduling servers group the tasks by the executing node, it gets easier and faster to manipulate them. Of course, before doing anything with those tasks, the third phase algorithm must make sure that only the prepared tasks are being manipulated at the time.

When the tasks are grouped by the executing nodes, the tasks can be assigned to the scheduling servers by assigning the executing nodes to them. So when the scheduling servers are sending tasks to their children, they are actually sending the executing nodes for them to handle. This is done because connecting and disconnecting from the executing nodes is more expensive than executing the tasks on them in an attempt to minimize the number of connecting and disconnecting. In this way, by assigning the executing nodes to the scheduling servers, the scheduling servers have more parameters for assigning the executing nodes to each other and they have a lot less work. In other words, the servers need to assign only the executing nodes and not all the tasks one by one.

There is also one additional problem that needs to be addressed in this approach. Only the tasks that are prepared can be assigned to the scheduling servers and they have to wait on the root scheduler for that moment. So when a task is prepared, it can be sent to another scheduling server or be assigned to the current one. The question is when to send the new tasks to the scheduling servers. It can be done at the moment they become prepared or a bit later in a batch. If the prepared tasks are sent as soon as possible, they will be available to the scheduling servers to schedule them sooner. However, if they are available later, it will not be as expensive to send them more than to send single tasks. The even bigger problem is that, if the tasks were sent one by one, the evaluation would not be able to finish in real time (it would take a lot longer than one hour). For this reason, a new variable percentage (later just *p*) has been used. When each scheduling server finishes scheduling *p* percent of its assigned tasks, then it signals to get more tasks.

Pseudocode of the algorithm used in this phase can be found in Algorithm 4-8.

```
// root scheduling server only
main (tasks) {
        while (have tasks) {
                prepared_tasks = get_prepared_tasks(tasks)
                schedule(prepared_taks)
        }
        finish_receiving_tasks()

        time = moment when last task was finished with execution

        return time
}


// all remaining servers
main() {
        tasks = receive tasks from parent
        while(1)
                IN PARALLEL {
                        schedule(tasks)
                        send results to parent
                        tasks = receive tasks from parent
                }
}


schedule(tasks) {
        grouped_tasks = group tasks by executing nodes;

        for child in children {
                child_tasks = select tasks for nodes that are assigned to
                                that child // heuristic
                send child tasks to child to schedule p tasks
        }

        local_tasks = tasks for nodes that are assigned to self

        schedule p tasks // algorithm from second phase
}
```

Algorithm 4-8 Third phase, Algorithm 8

4.8 The Third Phase Performance on Example Dataset

Since the third phase is the phase where everything kind of comes together, there are bound to be more tests than in the first two phases. More about those tests will be mentioned in the next chapter. Some combinations that were created during the third phase of the testing are presented in this section. The main new algorithm for this phase is described in the previous section. All the other second phase algorithms that were used in this phase are also described in this chapter. Table 4-6 presents how those combinations that are selected for display evaluate against the example dataset. Those exact results are chosen to demonstrate the effectiveness of the hand

and the machine learned heuristics (from the second and third phase) on different second phase algorithms.

Table 4-6 The third phase approaches on the example dataset

Phase 3	3. Hand heur, Alg 3	3. Hand, Alg 5	ANN, Alg 3	ANN, Alg 5
F node, 1 hand task heur, 3 servers	2460	3660	2450	3700
F node, 1 hand task heur, 7 servers	1270	3670	3650	3670
CC task & node heur, 3 servers	2460	3660	2450	3650
CC task & node heur, 7 servers	1270	3670	2470	3700

In this table, one column represents a combination of the third phase heuristic with the second phase single scheduling server algorithm. Each row is a combination of the executing node heuristic, the task heuristic and the number of the scheduling servers in the current system. The third phase heuristic (*3. hand* heuristic) is described in the third chapter. Those ANN heuristics are machine learned artificial neural networks. Different weights are used for each of those table elements. Those weights were learned using the second phase algorithm that stands next to them in the table column. They were learned by using the element's row as well. More precisely, those node and task heuristics and the number of the servers that are in that row. Those cooperative coevolution (CC) task and node heuristics are heuristics that were learned together in the second phase. They were learned using the second phase algorithms 3 and 5. The elements in the table are created by using those learned heuristics and algorithms accordingly.

Since this dataset is pretty small for *3. hand* heuristic, it seems it does not matter which task and executing node heuristics are used. This is probably because every scheduling server can now open two connections and there are only four executing nodes. Here it seems that the learned heuristics are not that successful at scaling to the example dataset. But it still seems that they are producing the best results for 3 scheduling servers (CC, ANN, algorithms 3 and 5).

Some images showing Gantt charts how this multi-server scheduling system schedules the tasks from the example dataset using the approaches from the third phase are below.

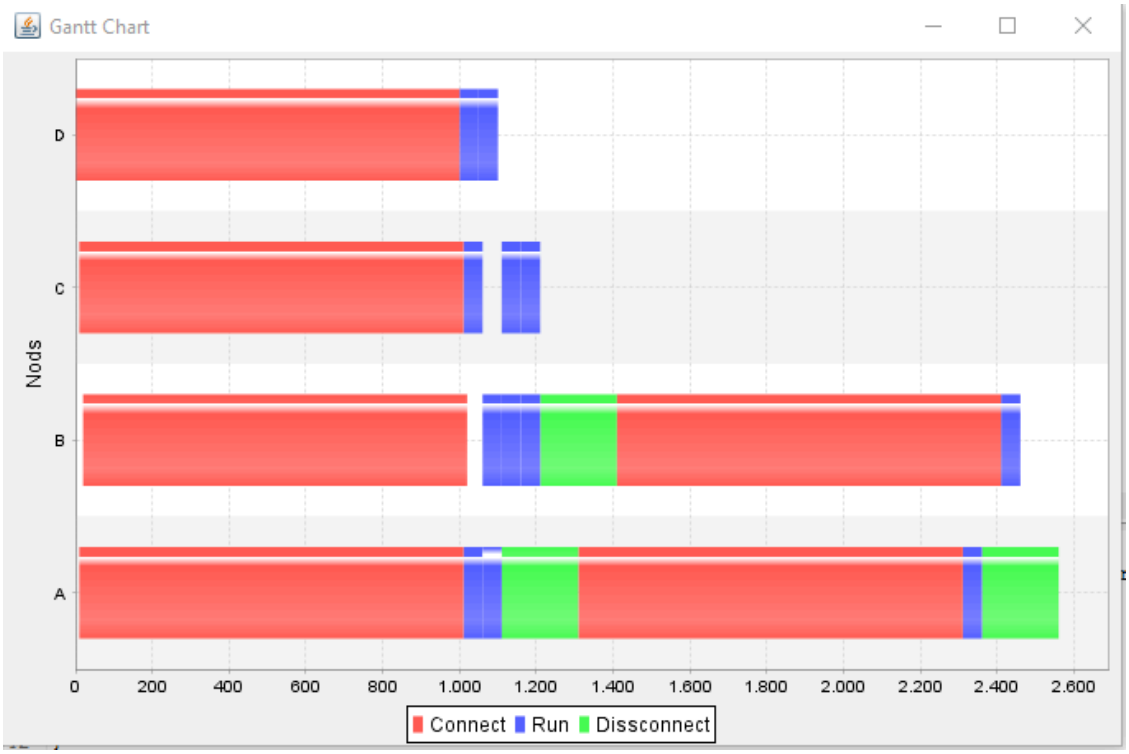

Image 4-13 Algorithm 3, 3. hand server heuristic, 3 scheduling servers, F node, 1. hand task heuristic and CC node, task heuristic

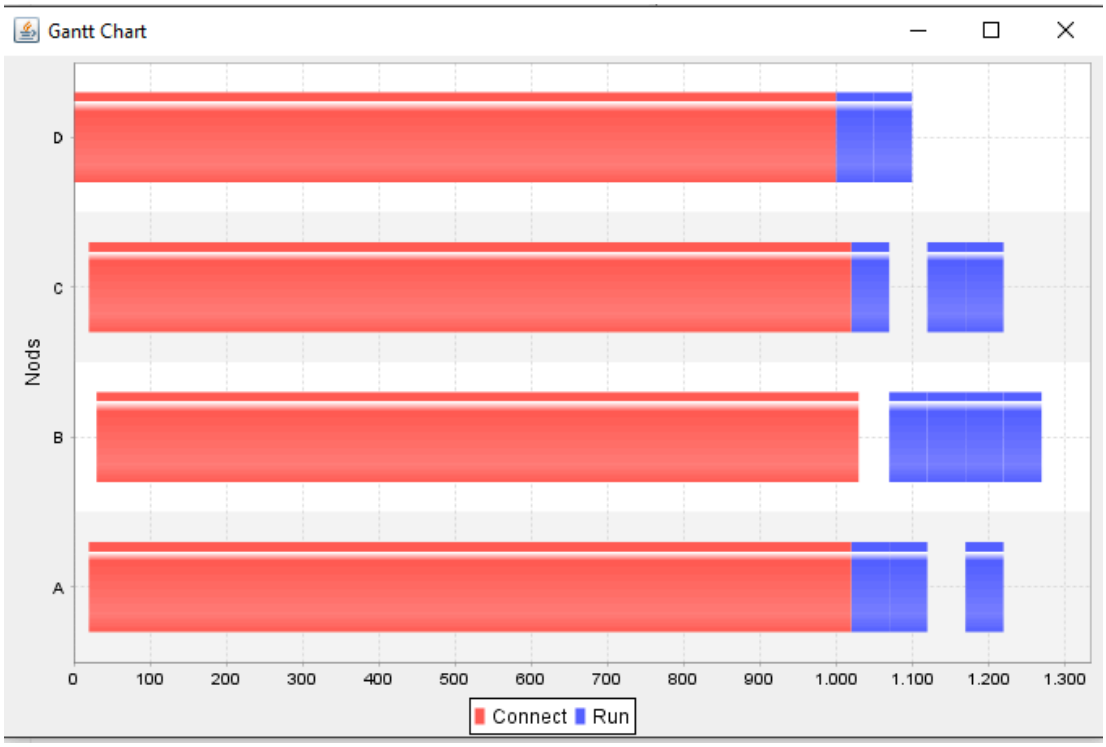

Image 4-14 Algorithm 3, 3. hand server heuristic, 7 scheduling servers, F node, 1. hand task heuristic and CC node, task heuristic

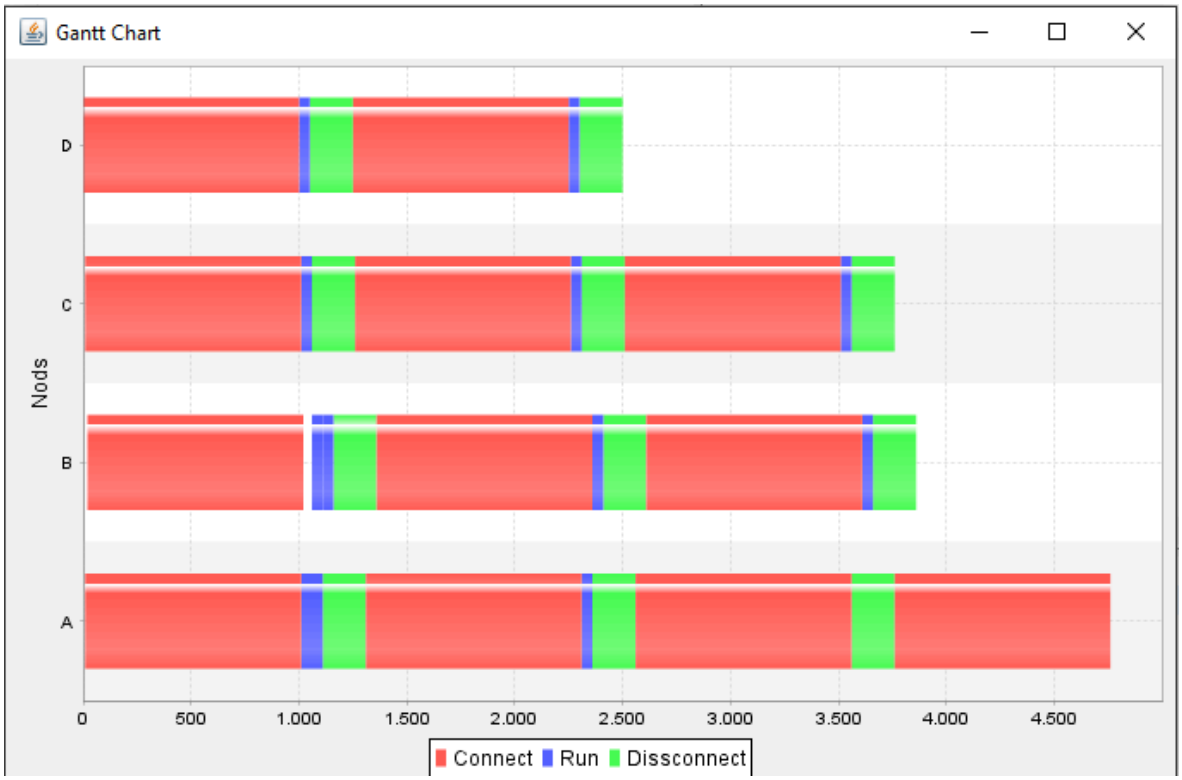

Image 4-15 Algorithm 5, 3. hand server heuristic, 3 scheduling servers, F node, 1. hand task heuristic and CC node, task heuristic

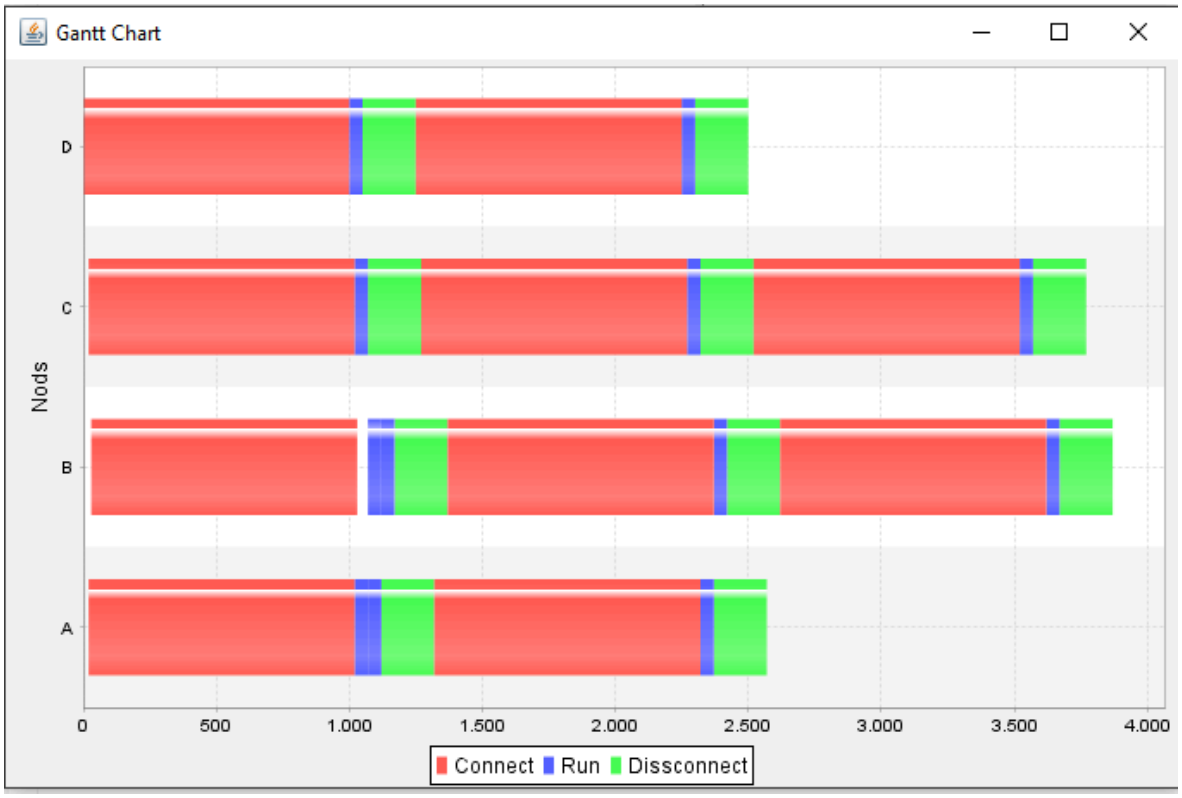

Image 4-16 Algorithm 5, 3. hand server heuristic, 7 scheduling servers, F node, 1. hand task heuristic and CC node, task heuristic

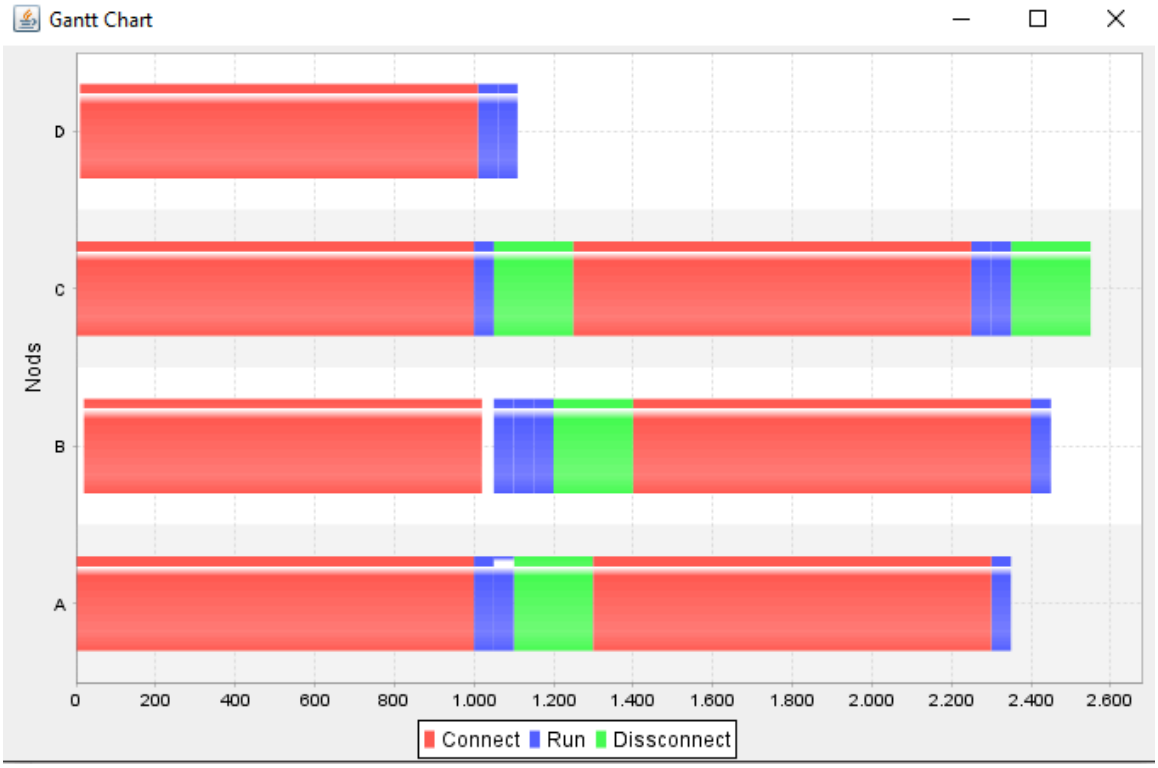

Image 4-17 Algorithm 3, ANN, 3 scheduling servers, CC node and task heuristic

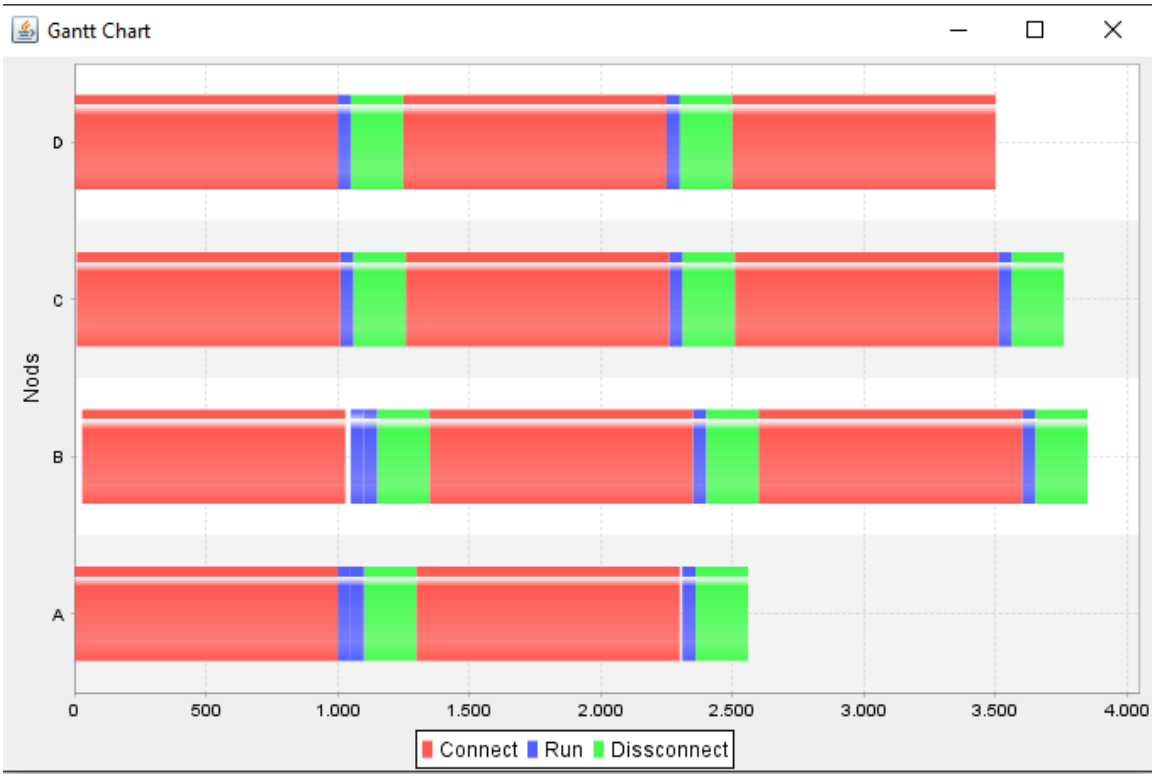

Image 4-18 Algorithm 3, ANN, 3 scheduling servers, CC node and task heuristic

5. Results

This chapter presents some results that were created while evaluating and testing the designed systems. Those results are composed on the smaller version data load of tasks (around 10 thousand tasks) and on the full load (around 3 million tasks). Some results are from executing evaluation with the smaller version and some are from the full load of tasks. All machine learning heuristics were trained using only the smaller version since it would take a few weeks to do it with the full load.

This chapter is also divided according to the testing phases. In each section there are presented some of the hand written heuristics and machine learned heuristics in combination with that section's approaches. Here one can see how effective hand written and machine learned heuristics really are.

All results presented in this chapter are from running on notebook computer that uses AMD A10-575 M (quad core, 2.5 GHz) and 12 GB DDR3L SDRAM. All algorithms, including heuristics, were coded in Java and run using JRE 8.

5.1 The First Phase

In this section the results from executing the evaluations with the approaches from the first phase are shown. Since it would take around two to three days to execute the full load with the approach from the first phase, all the results from this phase are only presented in the smaller version task load (around 10 thousand tasks). In Table 5-1 one can see how each task heuristic from the first phase and the method for selecting the nodes to disconnect (both described in the third and fourth chapter) perform on those tasks. For those combinations one can also see how long it takes to execute one evaluation.

Table 5-1 First phase approaches on smaller version dataset

Heuristics	Node closing method	Results	Evaluation time (s)
Max successors	FIFO	367050	461,10
Max successors	LRU	364650	599,59
Max successors	LFU	326500	513,42
Earliest ready	FIFO	1855450	208,45
Earliest ready	LRU	1269850	371,02
Earliest ready	LFU	328500	354,68
Execution tree rank	FIFO	1849750	438,87
Execution tree rank	LRU	1273050	392,94
Execution tree rank	LFU	289450	361,70
1. Task Hand	FIFO	285600	358,98
1. Task Hand	LRU	290900	358,06
1. Task Hand	LFU	246950	358,24

In the graph in Image 5-1, it can be seen how the heuristics and methods for selecting the nodes to disconnect performed in contrast to each other. Since the *earliest ready* and *executing tree rank* are kind of similar, their performance is very similar as well. Except using LRU, it seems that the *executing tree rank* is somewhat better than the earliest ready heuristic.

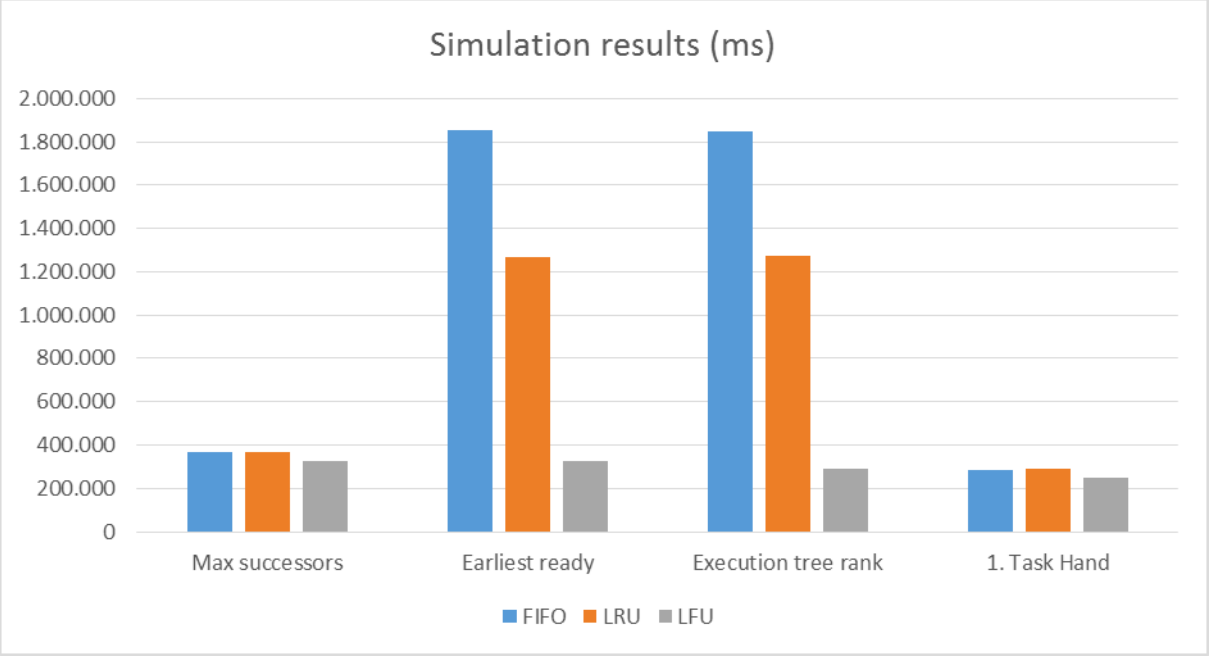

Image 5-1 Comparison of the first phase approaches' results on the smaller version dataset

As announced in the previous chapters, now it can be seen that the *1. task hand* heuristic outperforms all the other first phase heuristics. And it outperforms them using every method for selecting the nodes to disconnect. From the graph it may seem that the *max successors* is close to it, but from the Table 5-1 they are far away. Those two are also behaving kind of similarly. This is probably because the *1. task hand* heuristic contains max successors heuristic (more in the third chapter).

Image 5-2 shows the comparison of the evaluation times of those combinations of the heuristics and methods for selecting the nodes to disconnect. They all perform somewhat similarly due to all of them using the same algorithm. The fastest time being 208 sec (3.46 min) and the slowest time being 600 sec (10 min), the *1. task hand* heuristic that will be used in the next phases overall performs with 358 sec (5.97 min).

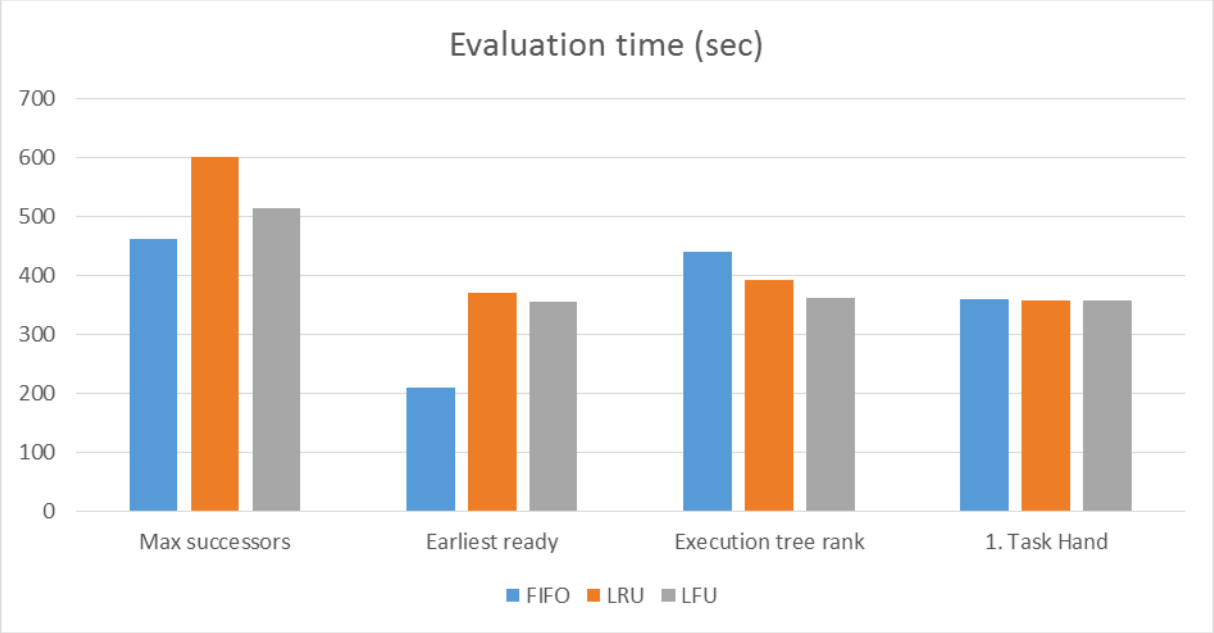

Image 5-2 Comparison of the first phase approaches evaluation times on the smaller version dataset

5.2 The Second Phase

In this section it is presented how well the approaches designed in the second phase perform on the full dataset (around 3 million tasks) and smaller version dataset (around 10 thousand tasks). All the heuristics that were hand created for scheduling the executing nodes are also presented here. With those hand heuristics for scheduling, only *1. hand task* heuristic was used for scheduling the tasks. Designed algorithms are described in the fourth section and hand created heuristics are presented in the third section. The task and executing node heuristics that are machine learned using this smaller version dataset are also presented here. All of those are presented in Table 5-2.

In this table, wherever CC appears, it denotes cooperative coevolution and represents one exact pair of the executing node and task heuristic that were learned using that row's algorithm. This means that the learned heuristics for algorithm 3 and 4 are not the same heuristics but different ones, where heuristics paired with algorithm 3 was learned using algorithm 3 and the pair with algorithm 4 is the one that was learned using algorithm 4.

Table 5-2 Second phase approaches on the small version and full (big) dataset

Algorithm	Heur nodes	Heur tasks	Result small	Evaluation small (s)	Result big	Evaluation big (s)
2	F	1. hand	2405000	532,480	---	---
2	G	1. hand	1921950	967,427	---	---
2	H	1. hand	19387650	1153,649	---	---
3	F	1. hand	1300850	4,826	4456700	126,305
3	G	1. hand	1378150	0,797	4774900	1927,113
3	H	1. hand	336500	0,878	4499500	2605,064
3	CC	CC	249900	42,754	4196650	3071,647
4	F	1. hand	1312250	4,019	4457700	160,598
4	G	1. hand	1388350	0,838	4774950	1577,458
4	H	1. hand	336500	0,908	4500500	1860,851
4	CC	CC	258950	39,823	4325950	2991,878
5	F	1. hand	334850	0,760	4185700	104,653
5	G	1. hand	334750	56,663	4185750	2760,148
5	H	1. hand	242750	18,493	4187300	2344,396
5	CC	CC	242000	46,953	4180900	2482,136
6	F	1. hand	1240600	3,930	4514650	191,902
6	G	1. hand	1105200	21,076	4593050	1581,792
6	H	1. hand	266900	9,814	4431350	1565,404
6	CC	CC	252050	46,088	4725050	2599,719
7	F	1. hand	1248650	4,063	4514650	180,669
7	G	1. hand	1112050	23,241	4593050	1746,873
7	H	1. hand	268000	10,066	4432350	1851,070
7	CC	CC	258500	43,937	5291300	2730,876

In this table one can see combinations of all the designed algorithms from the second phase with the executing node heuristics. Algorithm 2 does not have the results of the evaluation on the full dataset because it would take too long to execute. This can be seen from the evaluation times of the smaller version dataset. Algorithm 2 takes much longer to evaluate that set than other algorithms. The other algorithms seem to be evaluating similarly to one another. Their main difference in evaluation times is that some heuristics require parameters that take more time to calculate. And since the machine learned heuristics use all the parameters, it often takes the longest to calculate them.

From graph in Image 5-3 one can see the results of the listed algorithm heuristics pairs as they perform on the smaller version dataset. Since the machine learned heuristics were learned by using this dataset, it is expected that they outperform all the other hand written heuristics. They do so for every algorithm individually and through that they hold the best solution so far on this small dataset. From the hand written heuristics it seems that the heuristic H outperforms all other hand written heuristics.

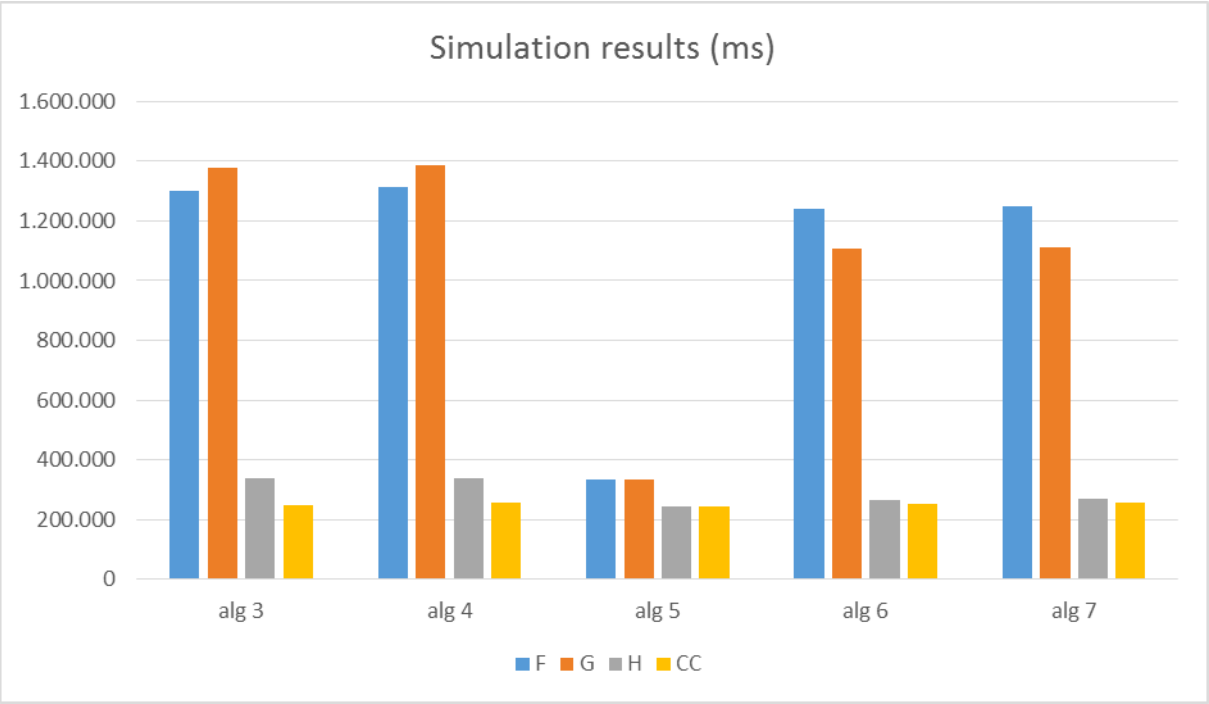

Image 5-3 Comparison of the second phase approaches' results on the smaller version dataset

Graph in Image 5-4 presents the comparison of the evaluation times of the presented algorithm heuristic approaches. As mentioned earlier, the machine learned heuristics use all the available parameters, so it is expected to last longer to evaluate them than other heuristics. This is not so only for algorithm 5, but since this is a real time, not a thread time, this difference might not be so big. Overall, the fastest executing heuristic could be F but it presents the worst results.

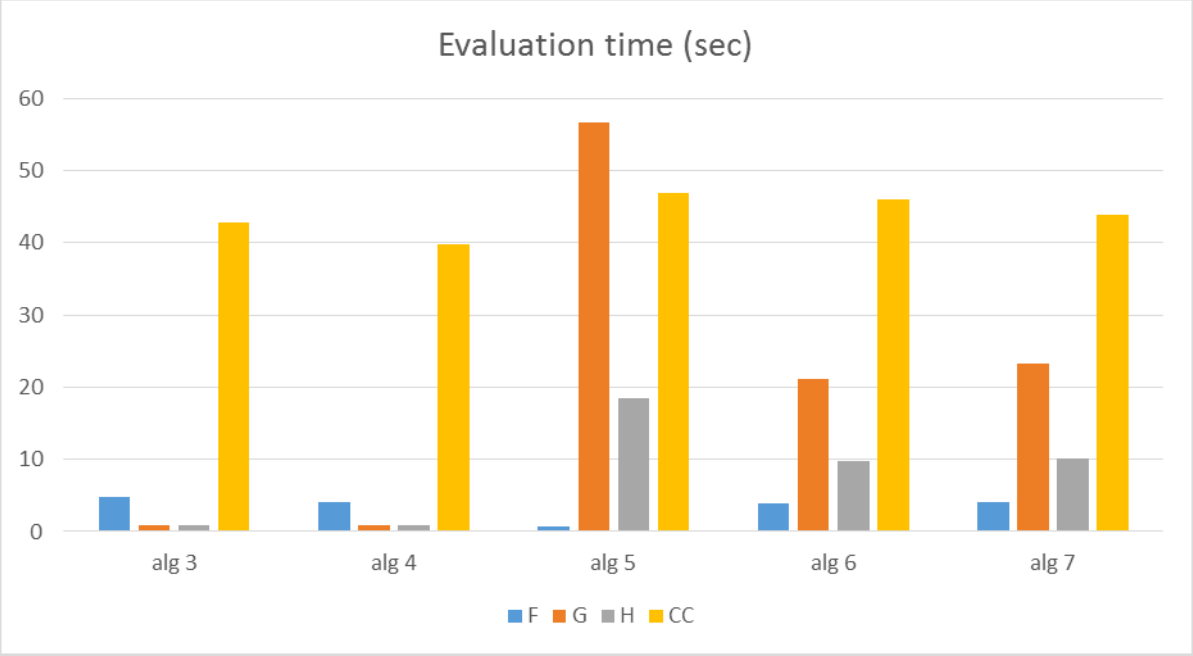

Image 5-4 Comparison of the second phase approaches evaluation times on the smaller version dataset

Image 5-5 represents the results of the same heuristics only when applied to the full dataset. It might seem that for all the algorithms and heuristics the results are more similar, but that is mostly because the scale is a lot bigger. The biggest question here is how the machine learned heuristics performed on this new larger dataset they were not learned on. For three out of five algorithms they produced the best results but for the other two they produced the worst. They hold the best overall results with algorithm 5 (4180900).

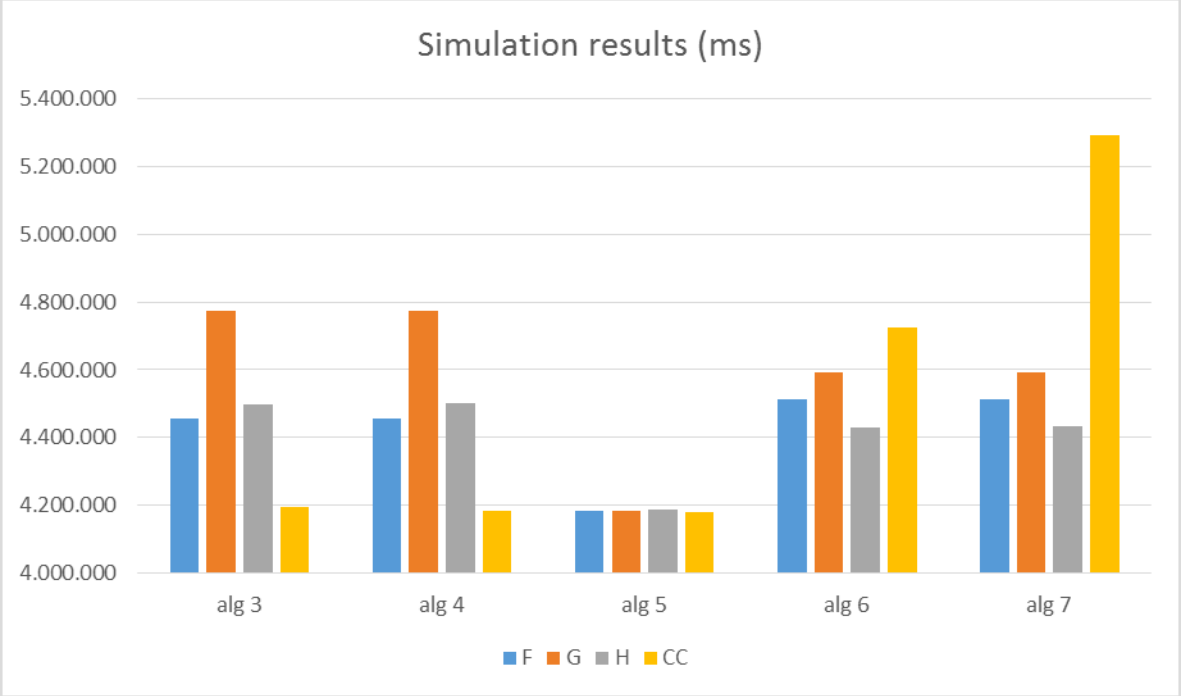

Image 5-5 Comparison of the second phase approaches' results on the full dataset

Image 5-6 is the graph of the evaluation times that it took for combinations of the algorithms and heuristics to perform on the full dataset. As expected, similar to the evaluations on the smaller version dataset, the machine learned heuristics took the longest to evaluate. On the other side, F heuristic took the least amount of the time to evaluate due to the time needed to calculate all the parameters needed for each heuristic.

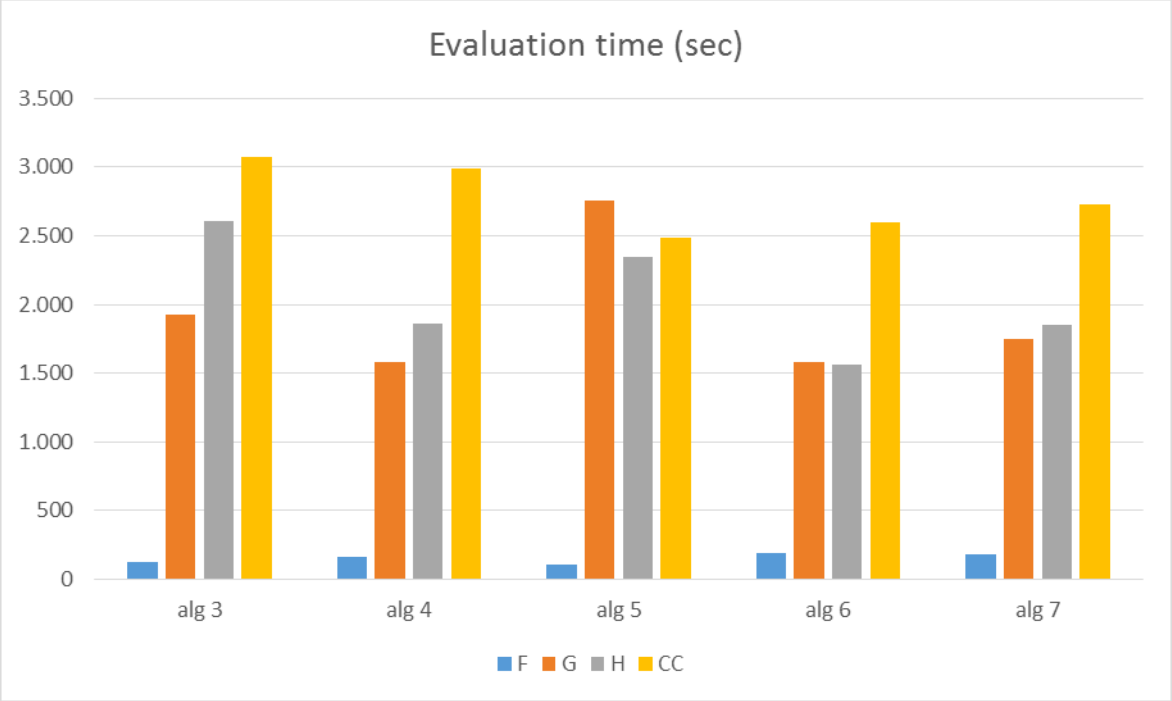

Image 5-6 Comparison of the second phase approaches evaluation times on the full dataset

5.3 The Third Phase

The third phase is the phase that removes one scheduling server simplification. By doing this, it adds more parameters to the system. The number of the scheduling servers and heuristics to assign the executing nodes (by assigning tasks) to the scheduling servers is most important. Since this is the final phase, a lot of tests were done in it. Most of it can be found in the CD attached to this thesis.

The heuristics learned in this phase were also learned by using the smaller version dataset. The main idea here was to test and to compare the effectiveness of different heuristics combinations. A lot of hand and learned heuristics were combined and tested on both the smaller and the full version dataset. Those results were then compared in a way to see whether they scale. In other words, the goal is to see whether one combination is better than some other on the smaller dataset and whether it remains better on the full dataset. All of those comparison results can be found in the attached CD and will be discussed in the next chapter.

The results presented here are used to demonstrate the effectiveness of the system under different second phase algorithms, second phase and third phase heuristics. Only *1. hand task* heuristic is used from the first phase. There are multiple hand written and machine learned heuristics from the second phase. There are also multiple single scheduler algorithms that were created in the second phase and used here. The machine learned heuristics for tasks and executing nodes were learned during the second phase and used only during the third phase. There is only one hand heuristic from the third phase, which is described in the fourth chapter. The results from this phase are separated into two main categories. In one category the

system has 3 and in the other the system has 7 scheduling servers. The results from the system that has 3 scheduling servers are presented in Table 5-3, while the results from the system with 7 scheduling servers are presented in Table 5-4.

The task and executing node heuristics with cooperative coevolution (CC) are the same as those from the previous section (the second phase). They represent two heuristics that were learned together using their row's algorithm during the second phase. Machine learned scheduling server heuristics, those artificial neural networks (ANN) are learned using the system that corresponds to the parameters in its row. For example, weights for ANN in the third row of Table 5-3 are learned by using the system with three scheduling servers, algorithm 3 for single scheduling server, heuristic F for executing nodes and 1*. hand task* heuristic for tasks.

Table 5-3 The third phase approaches on the small version and the full (big) dataset using 3 scheduling servers

Algorithm	Node heur	Task heur	Server heur	Results small	Evaluation small (s)	Results big	Evaluation big (s)
3	F	1. hand	3. hand	229100	2,777	2721550	127,393
5	F	1. hand	3. hand	94184	1,855	1504600	83,755
3	F	1. hand	ANN	154300	1,749	2373980	80,049
5	F	1. hand	ANN	93534	1,784	1545050	82,964
3	CC	CC	3. hand	196434	3,741	4410611	890,897
5	CC	CC	3. hand	93550	7,723	1504600	1570,035
3	CC	CC	ANN	155181	12,324	1924200	2930,515
5	CC	CC	ANN	93234	12,944	1543450	2644,124

Table 5-4 The third phase approaches on the small version and the full (big) dataset using 7 scheduling servers

Algorithm	Node heur	Task heur	Server heur	Results small	Evaluation small (s)	Results big	Evaluation big (s)
3	F	1. hand	3. hand	378634	2,241	2625950	113,642
5	F	1. hand	3. hand	93450	1,572	1504650	66,632
3	F	1. hand	ANN	118483	1,849	3020995	91,932
5	F	1. hand	ANN	49718	1,722	1308650	75,379
3	CC	CC	3. hand	346234	4,842	1764850	1244,750
5	CC	CC	3. hand	94550	9,834	1504600	1653,355
3	CC	CC	ANN	160804	13,248	2557107	2637,239
5	CC	CC	ANN	48820	13,151	959250	2342,740

In this table one can see some combinations of the designed algorithms from the second phase with multiple different heuristics. Only the second phase algorithms 3 and 5 were used here for learning scheduling server heuristics. The reason for that is that those two algorithms showed most consistency with machine learned heuristics in the second phase. To be more exact, since all the learned second phase heuristics (CC for tasks and executing nodes) were learned on the smaller version dataset, only the heuristics for those two algorithms outperformed other heuristics on the smaller dataset. More importantly, they also outperformed other heuristics on both the example and the full dataset.

The graphs in Image 5-7 and Image 5-8 present the results of the evaluation against the smaller version dataset using 3 and 7 scheduling servers, respectively. Those results are composed of the second phase algorithms, second and third phase heuristics. Since the third phase learned heuristics were learned adjacent to the second phase algorithms and heuristics on this dataset, it is expected they outperform their hand written counterparts.

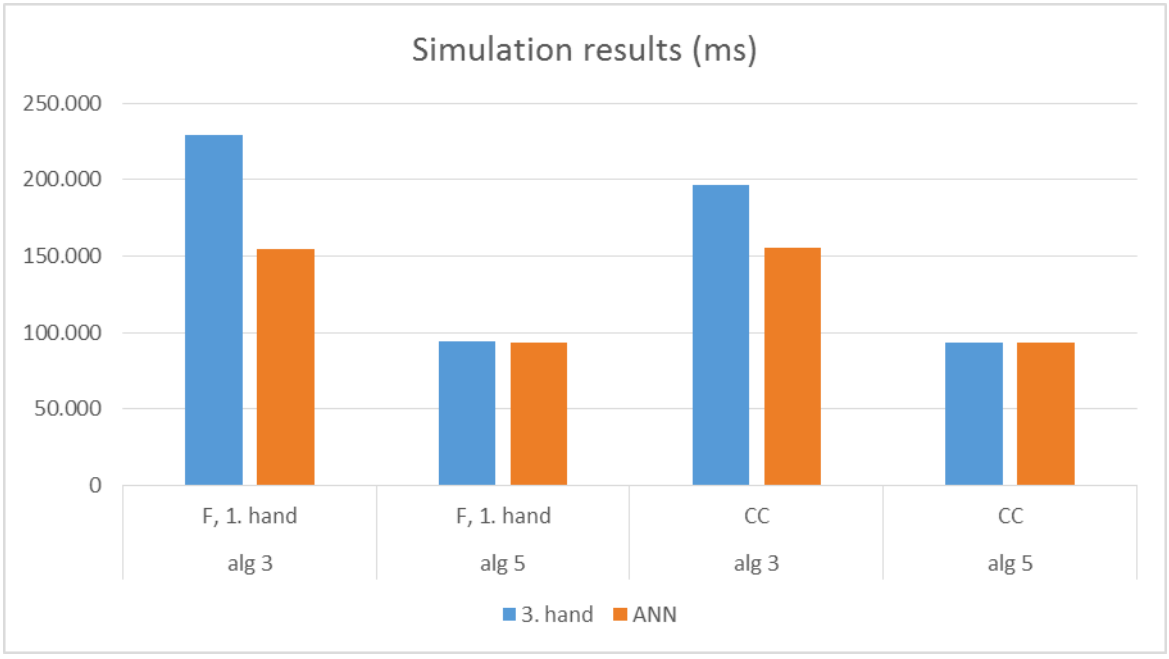

Image 5-7 Comparison of the third phase approaches results on the smaller version dataset using 3 scheduling servers

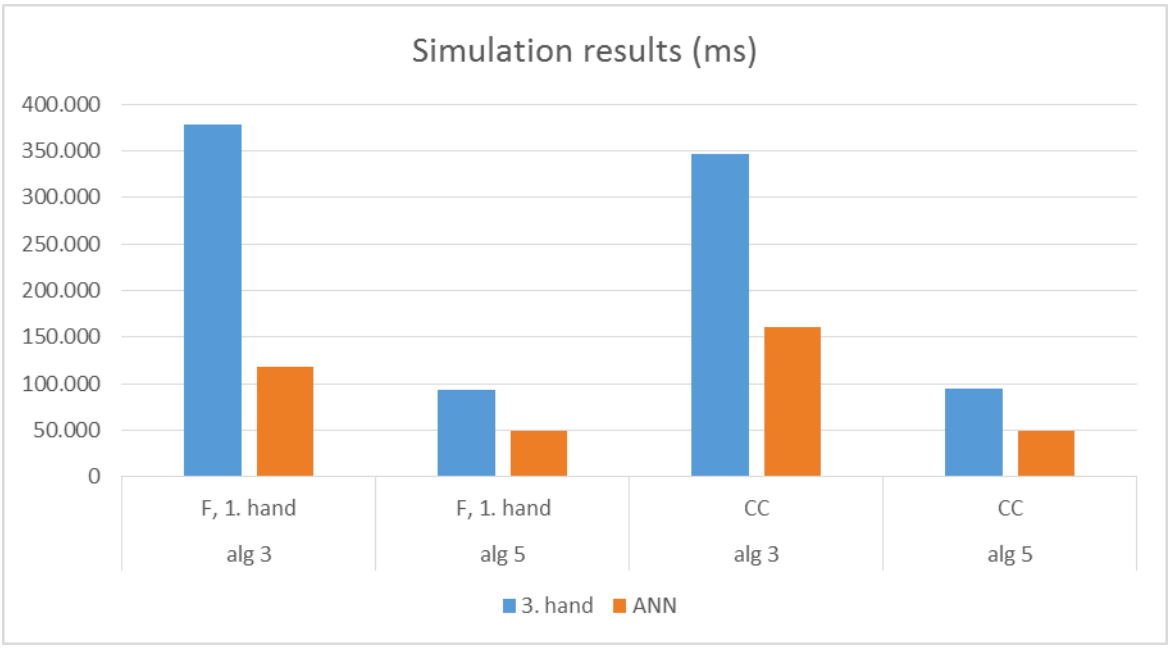

Image 5-8 Comparison of third phase approaches results on smaller version dataset using 7 scheduling servers

Image 5-9 and Image 5-10 present the comparison of the evaluation times of the presented third phase approaches on the systems with 3 and 7 scheduling servers, respectively. As mentioned in the previous section, since machine learned heuristics use all available parameters, it is expected for them to need more real time to finish

the evaluation than the other heuristics. It seems like this is true for the third phase heuristic that was learned using CC, but not the ones that were learned using F node heuristic and *1. hand task* heuristic. Here it seems that best results are produced by learned third phase heuristic using algorithm 5.

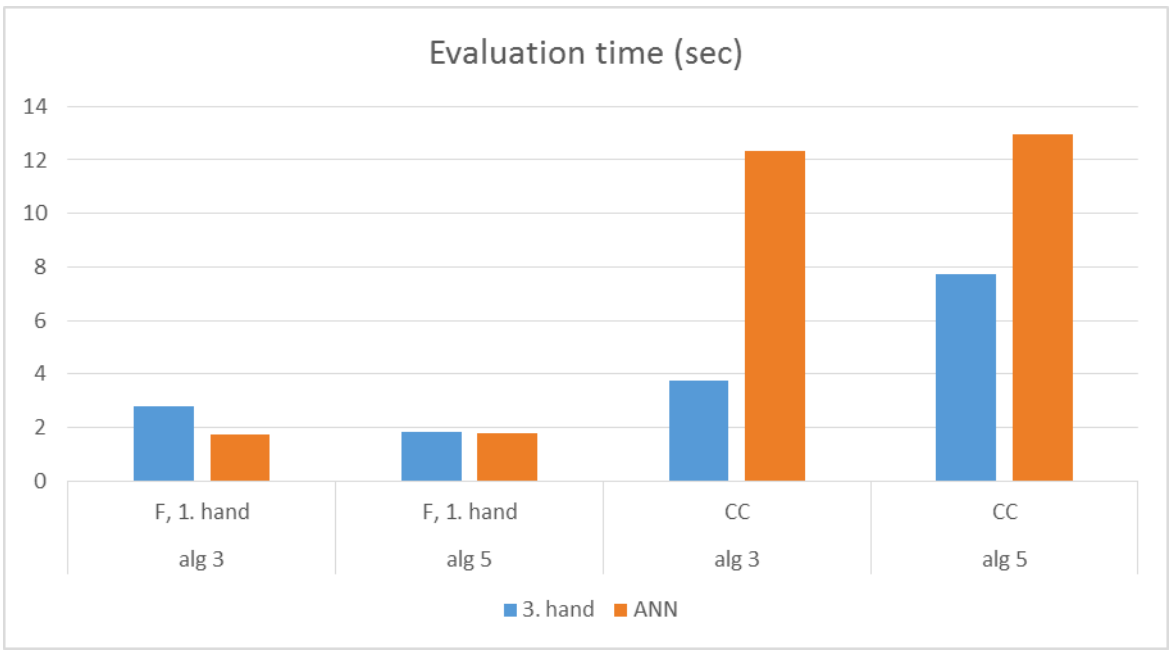

Image 5-9 Comparison of the third phase approaches evaluation times on the smaller version dataset using 3 scheduling servers

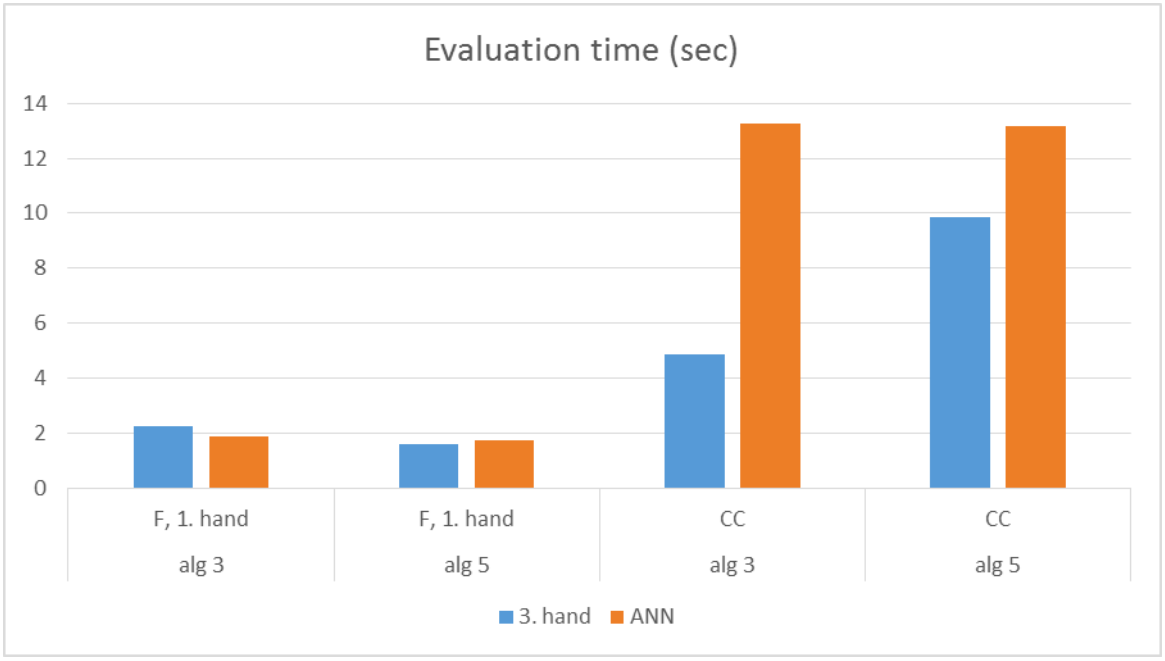

Image 5-10 Comparison of the third phase approaches evaluation times on the smaller version dataset using 7 scheduling servers

Graphs in Image 5-11 and Image 5-12 present the results of the evaluation against the full dataset using 3 and 7 scheduling servers, respectively. Here it seems that the learned heuristics outperform hand heuristics using algorithm 3 with 3 scheduling servers and algorithm 5 with 7 scheduling servers. Those two combinations seem to scale great from the smaller version dataset. But it seems that algorithm 5 with 3 scheduling servers as well as algorithm 3 with 7 scheduling servers outperform the learned heuristics while using the hand written one. Overall, the best score here is produced by using the third phase learned heuristic with algorithm 5 and the learned executing node and tasks heuristics from the second phase on 7 scheduling servers.

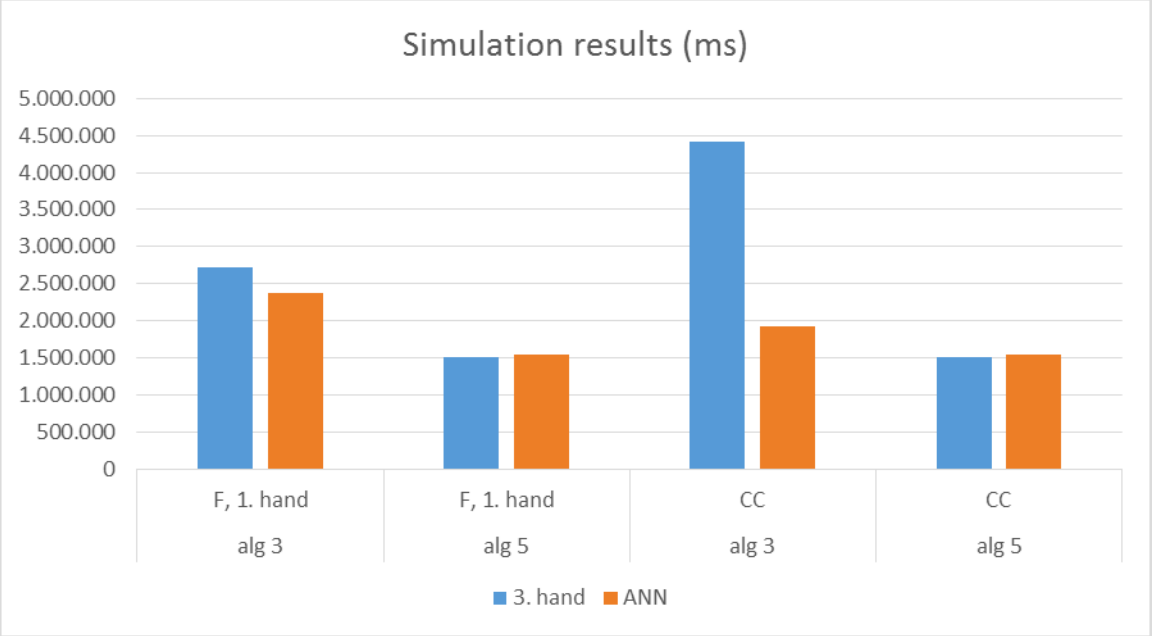

Image 5-11 Comparison of the third phase approaches results on the full dataset using 3 scheduling servers

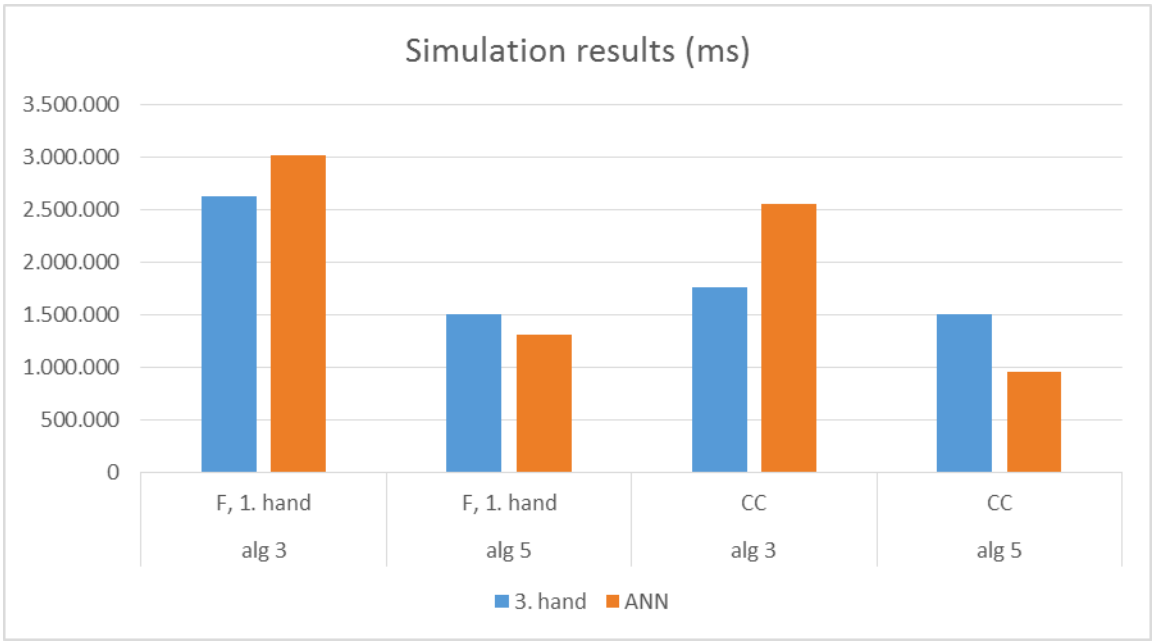

Image 5-12 Comparison of third phase approaches results on the full dataset using 7 scheduling servers

Image 5-13 and Image 5-14 present the comparison of the evaluation times of the presented third phase approaches on systems with 3 and 7 scheduling servers, respectively. Evaluation times presented here are durations needed to evaluate on the full dataset. From those graphs it seems that the learned second phase heuristics

are the main ones responsible for slowing the evaluation. It also seems that the learned heuristics from the third phase do not have that much effect on the evaluation time.

As mentioned in the previous section, since the machine learned heuristics use all the available parameters, it is expected of them to need more real time to finish the evaluation than the other heuristics. It seems like this is true for the third phase heuristic that was learned using CC, but not the ones that were learned using F node heuristic and the *1. hand task* heuristic. Here it seems that the best results are produced by the third phase learned heuristic using algorithm 5. For the hand written node and task heuristics it seems to be faster using the learned scheduling server heuristic but for the learned node and task heuristics it is faster to use the hand written one.

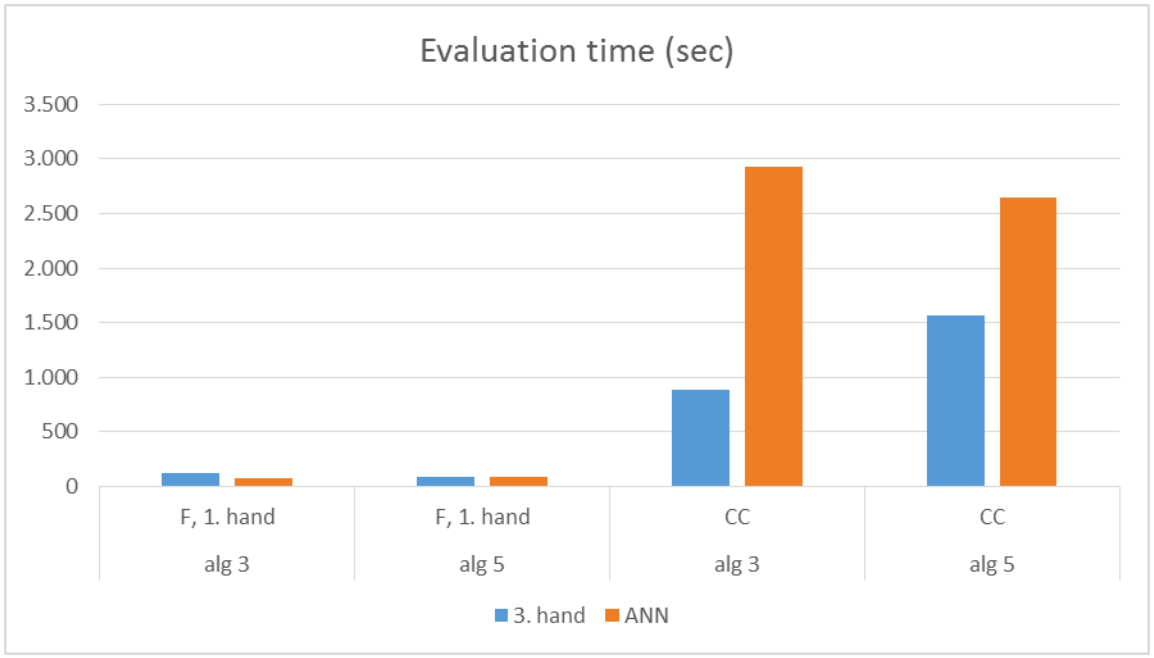

Image 5-13 Comparison of third phase approaches evaluation times on the full dataset using 3 scheduling servers

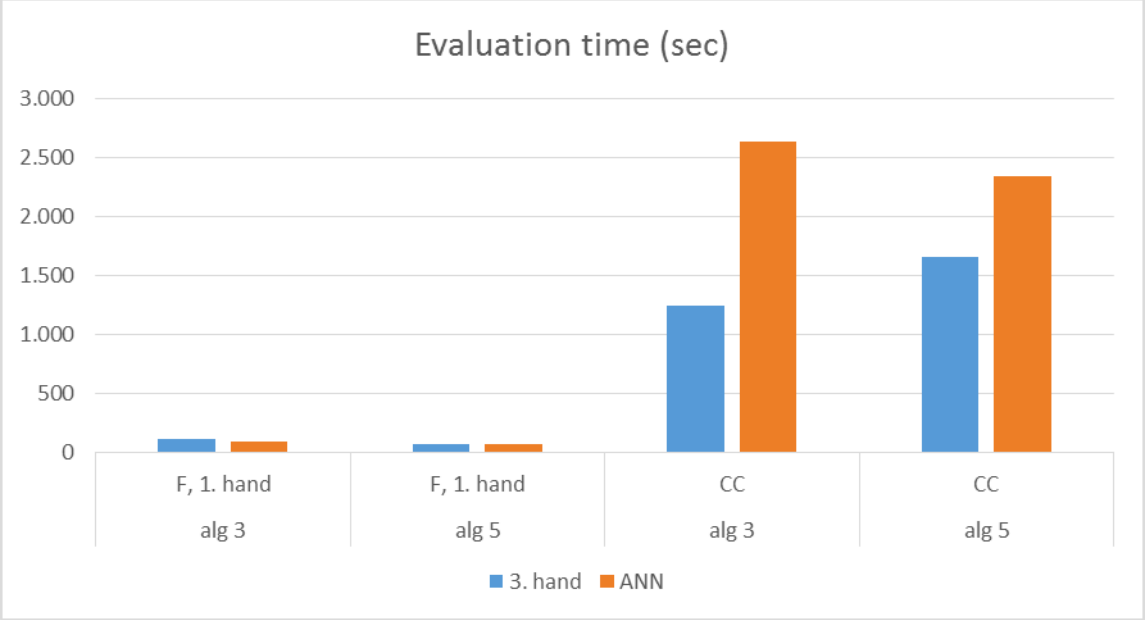

Image 5-14 Comparison of third phase approaches evaluation times on the full dataset using 7 scheduling servers

5.4 Top Results

As mentioned in the previous chapters, there are a lot of combinations of the possible task, executing node and scheduling server heuristics. There are also several algorithms developed for scheduling that use those heuristics. To put it more in perspective, every scheduling server heuristic that was learned using some combination of a task and node heuristic can be evaluated using some other combination of a task and node heuristic. The problem with this is that the combinations do not always scale in performance from the smaller version dataset to the full dataset. In other words, if combination A is outperforming combination B on the smaller version dataset, combination B can still outperform combination A on the full dataset.

The number of overall possible combinations that are presented is measured in thousands. It would take too long to test them all on the big dataset. And also the idea here is to find a combination with an overall great performance, not the one that just fits the full dataset, so it could be used in real systems.

To find a good combination, a lot of combinations were first tested on the smaller version dataset (around 4,500 combinations). Then the best 200 combinations were chosen for the second round. For the second round, the full dataset was split into multiple pieces in size of the smaller version (around 300 pieces). Those 200 best combinations were then evaluated on 20 pieces from the full dataset. Every combination (C_i) was then averaged using Equation (5.1) where $Part_j$ is one of those 20 pieces and an RC_i ($Part_j$) is an evaluation result of combination i against part j.

$$\frac{\sum_{j=1}^{n} RC_i(Part_j)}{n} \quad (5.1)$$

From those 200 averaged results, 50 best combinations were chosen to evaluate against the full dataset. They were ranked based on their performance (that is, the better combination has a higher rank). Primarily, they are based on those averages and secondly on their results against the full dataset. Those ranks were then compared and combination's ranked position changes have been calculated. In other words, if one combination was at rank 3 on the averages and rank 5 on the full dataset, the ranked position change would be of -2.

In Table 5-5 the best five results for averages are presented. Those are results for 7 servers only since 7 servers outperform 3 servers. With node and task heuristics where it says CC 5, it means they were trained as cooperative coevolution with algorithm 5. Similarly learned server heuristics are represented as NN <algorithm> <executing node heuristic> <task heuristic> <number of servers>. Along with those heuristics, their average results and the results on the full dataset are also presented. With those values, their rank position changes are also presented.

Table 5-5 Best five from averages

Algorithm	Node heur	Task heur	Server heur	Results avg	Results big	Rank change
5	CC 5	CC 5	NN 5 CC CC 7	30082	959250	0
5	CC 5	CC 5	NN 5 F 1.hand 7	30094	1297500	0
5	G	CC 3	NN 5 F 1.hand 7	30151	1301650	-24
5	F	1. hand	NN 5 F 1.hand 7	30370	1301850	-26
5	G	1. hand	NN 5 F 1.hand 7	30863	1301800	-24

6. Conclusion

A special case of makespan optimization in scheduling is tackled in this thesis. The problem consists of multiple different elements that need to be scheduled. The thesis presents the system model that fits the given problem. That system model was designed in three phases. The hardest constraint for the model to work was to run the evaluation with the full dataset in less than an hour.

The complete models consist of two main algorithms and three types of heuristics. The first algorithm is responsible for distributing tasks between scheduling servers. The second one is responsible for executing tasks on executing nodes from one scheduling server by connecting and disconnecting the executing nodes to that server. The heuristics can be divided in task heuristics, executing node heuristics and scheduling server heuristics. Scheduling server heuristics are responsible for assigning the executing nodes to the scheduling servers. Executing node heuristics are responsible for selecting which nodes to connect to and which nodes to disconnect from the scheduling server. Finally, task heuristics are responsible for selecting which tasks to execute next.

The thesis presents some of the possible greedy hand written heuristics which cover those functionalities. The thesis also presents some machine learning techniques that were used to improve the hand written heuristics. Those techniques include using genetic programming with symbolic regressing trees that also have some decision tree components in cooperative coevolution. Those were used to learn the executing node and task heuristics. Artificial neural networks trained by particle swarm optimization were used for scheduling server heuristics.

The results indicate that using machine learning techniques can greatly improve the system's efficiency. This presents a segment that can be further explored in order to produce even better system results than the ones that are presented in this thesis. The thing to note here is that by adding more parameters to the learning models, the models may produce better results but may take more time to produce those parameters. The hardest constraint states that the evaluation must be completed in less than an hour.

7. References

[1] Pinedo, M. L. Scheduling: Theory, Algorithms, and Systems. Third edition, NewYork, 2008.

[2] Scheduling (computing), https://en.wikipedia.org/wiki/Scheduling_(computing), May 2016.

[3] Coulouris, George; Jean Dollimore; Tim Kindberg; Gordon Blair. Distributed Systems: Concepts and Design (5th Edition). Boston, 2011.

[4] Genetic programming, http://en.wikipedia.org/wiki/Genetic_programming, May 2016.

[5] Koza, J. R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: The MIT Press. 1992.

[6] The GP Tutorial, http://www.geneticprogramming.com/Tutorial/, May 2016.

[7] James C. Bean: Genetic Algorithms and Random Keys for Sequencing and Optimization, University of Michigan 1993

[8] Coevolutionary algorithms, http://wiki.ece.cmu.edu/ddl/index.php/Coevolutionary_algorithms, May 2016.

[9] M. A. Potter and K. A. D. Jong, “A cooperative coevolutionary approach to function optimization,” in PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature. June 2005

[10] Kolmen D: Koevolucijski algoritmi. Faculty of Electrical Engineering and Computing, May 2011

[11] Artificial neural network, http://en.wikipedia.org/wiki/Artificial_neural_network, May 2016

[12] Marko Čupić: Prirodom inspirirani optimizacijski algoritmi. Metaheuristike, 2013.

[13] Activation functions, http://en.wikibooks.org/wiki/Artificial_Neural_Networks/Activation_Functions, May 2016

[14] Sigmoid function, http://en.wikipedia.org/wiki/Sigmoid_function, May 2016

[15] Čupić, Dalbelo Bašić, Gloub: Neizrazito, evolucijsko i neuroračunarstvo, 2013.

[16] Particle swarm optimization, https://en.wikipedia.org/wiki/Particle_swarm_optimization, May 2016

[17] Allaoua, Laoufi, Gasbaoui, Abderrahmani: Neuro-Fuzzy DC Motor Speed Control Using Particle Swarm Optimization, http://lejpt.academicdirect.org/A15/001_018.htm, May 2016

[18] Blum L. A., Langley P., Selection of relevant features and examples in machine learning. Pittsburgh, October 1995

[19] Glaubius R, Tidwell T., Gill C., Smart W. D. Real-Time Scheduling via Reinforcement Learning. Washington University in St. Louis 2010

[20] Shrivastava S. Reinforcement Learning for Scheduling Threads on a Multi-Core Processor, Stanford 2010

[21] Lee C.-Y., Piramuthu S., Tsai Y.-K. Job shop scheduling with a genetic algorithm and machine learning, Nov 2010

[22] Shaw M. J., Park C. S., R Narayan, Intelligent scheduling with machine learning capabilities: the induction of scheduling knowledge. Carnegie Mellon University 1990

On-line Scheduling Heuristics in Distributed Environments

Abstract

This thesis tackles a specific type of multiple machine scheduling problem. It deals with scheduling tasks on executing nodes through a network of scheduling servers, where the goal is to optimize the makespan. In this problem executing nodes represent the machines with eligibility restrictions. Tasks are machine bound because every task can only be executed at a specific machine and they can also be precedence constrained to other tasks. The thesis describes the tackling of the problem in three phases. Each of those phases presents a part of the system and some hand written heuristics that were used in order to solve the problem. The thesis also presents some machine learning techniques, like genetic programming and neural networks, which were used in order to produce the best possible heuristics.

Key words

Scheduling, multiple machines, scheduling heuristics, on-line execution, distributed environment, resource constrained, machine eligibility restrictions, machine learning techniques, genetic programming, artificial neural networks

Heurističko raspoređivanje na zahtjev u raspodjeljenoj okolini

Sažetak

Rad se bavi specifičnim tipom raspoređivanja na paralelnim strojevima. Bavi se raspoređivanjem poslova na izvršne čvorove kroz mrežu servera za raspoređivanje. Pri tome je cilj optimizacija vremena trajanja. U ovom problemu izvršni čvorovi predstavljaju strojeve sa ograničenjima pridruživanja poslova. Svaki posao je ograničen na samo jedan stroj, a početak izvođenja mu može ovisiti o nekom drugom zadatku. Rad opisuje tri faze rješavanja problema. U svakoj od faza je predstavljen dio sustava i neke rukom pisane heuristike koje su korištene u rješavanju problema. Rad isto predstavlja neke tehnike strojnog učenja poput genetskog programiranja i neuronskih mreža koje su korištene da bi se proizvele što bolje heuristike.

Ključne riječi

Raspoređivanje, paralelni strojevi, heuristike raspoređivanja, izvođenje u realnom vremenu, raspodjeljena okolina, ograničenja u resursima, ograničenja pridruživanja poslova, tehnike strojnog učenja, genetsko programiranje, suradnička koevolucija, umjetne neuronske mreže