Scheduling as a decision-making process plays an important role in most manufacturing and production systems. Most of the scheduling problems are complex, combinatorial and so difficult to solve [2010 dynamic]. The Job shop scheduling is a branch of schedule problems and it is well known as one of the most complicated optimization problems [ref]. Therefore, it was considerably difficult to obtain the optimal solution even for small-scale instances [2010 A hybrid]. For instance, the test problem (FT10) which consists of 10 Jobs and 10 Machines took researchers roughly 20 years to reach to optimality [ref 1 in 2010 A hybrid].

The Job shop scheduling problem (JSP) is to determine a schedule that aims to execute a set of predetermined jobs on a set of machines. More precisely, there are jobs to be processed on unrelated machines; then, in basic model of this general JSP, it is essential that each of these jobs require some number of operations in order to be completed. The routings of the jobs are deterministic and known a priori, as the processing time of each job on each machine. There are some assumptions to be followed:

1. No machine can process more than one operation at a time.
2. Each operation, once started, must be executed to completion.
3. Machines can never be broken-down and manpower of uniform ability is always available.
4. The job routing is given and no alternative routings are permitted.
5. All jobs are known and ready at time 0.

The main purpose is to determine in which time each operation should start to be executed while optimizing given objective(s). There can be several objectives associated with JSP; however, the most widely studies is minimizing makespan. Makespan means the time needed to complete/execute all the jobs in the floor.

Literature Review

As mentioned, the JSP is NP-Hard in nature. Therefore, there is no polynomial-time algorithm for this problem that guarantees an optimal solution. In recent years, many studies are being done toward the application of meta-heuristics for solving the JSP. Several meta-heuristic techniques such as genetic algorithm [ref], ant colony optimization [ref], particle swarm optimization [ref], Cuckoo Search algorithm [ref 2014 DCS], and Tabu Search [ref] have been developed and applied to solve JSP. Several dispatching rules have been proposed in [] to solve JSP. On the other hand, a classical depth-first branch and bound method are adopted to minimize the makespan of jobs based on the disjunctive graph model in [2019 BB using genetic prog].

The Job shop scheduling problem is one of the most complicated combinatorial optimization problems[ref]. It has been widely studied by many researchers over the past 60 years. The Job shop scheduling problem consists of many jobs that must be processed by a set of machines. In a typical shop manufacturer, each jab has several operations that must be done on a set of machines. The main idea is to construct a schedule that determines the execution of jobs on machines. However, a different sequence of jobs has been processed on machines will result in different production performance. The main goal is to search for the sequence of jobs that will provide the best shop floor performance. To achieve a better shop floor schedule, many researchers have developed several techniques to build a schedule that reaches the optimal solution. Since Job shop scheduling has been proved to be NP-hard, reaching the optimal solution has been proved extremely hard to reach, especially when the problem size is increased.

Many studies have developed effective methods for solving small-scale problems. However, there are relatively fewer studies on large-scale Job shop scheduling. (paper 2001) have proposed a heuristic that decomposes the problem into sub-problems (Time Window) and solving each sub-problem using a shift bottleneck heuristic while minimizing Total Weighted Tardiness. The decomposition method aimed to balance the number of operations per machine. The main drawback of this study is that it tackled a limited number of instances, and the makespan objective function was not analyzed. (Paper 2014) have introduced a decomposition heuristic based on multi-bottleneck machines to solve large-scale Job shop scheduling problems. The original problem was decomposed into sub-problems, and a Genetic Algorithm has been presented to solve each sub-problem. Instead of detecting the bottleneck machines as has been previously done, (paper 2008) has constructed a heuristic to obtain the characteristics values, including bottleneck jobs. Then, they adopt these characteristics values to coordinate the process of problem decomposition and the sub-problem solving where they have developed an adaptive Genetic Algorithm to solve each sub-problem.

Most of the current studies focused on identifying bottleneck machines/jobs and using incomplete optimization approaches to solve each sub-problem. In this work, we aim to show how the quality of the shop floor schedule will be affected when subproblems are solved using Answer Set Programming (ASP [ref]). Since the Job shop scheduling problem is complex, especially when the problem size is increased, decomposition methods have received attention from many researchers. These methods attempt to develop solutions to complex problems by decomposing a problem into a number of smaller sub-problems, which are traceable and easier to understand. Solutions are developed for each sub-problem individually and then integrated to form a solution to the original problem. Until now, there is no specific method to decompose a problem perfectly and reduce the mistakes – caused by the decomposition - as much as possible. This study introduces a new heuristic to decompose a whole problem into sub-problems, making a balance between Time Windows where the number of operations per each Time Window is the same. To decompose the problem, one must decide how to assign operations to each Time Window. The main idea of our proposed heuristic is to compute an estimated starting time for each operation. Where the estimated time of the first operations of all jobs is zero, and for the successive operations is the summation of the estimated starting time and the processing time of the previous operation.

The operations with the minimum estimated starting Time will be assigned to the first Time Window until reaching the maximum number of operations per time window. The rest of the unassigned operations will be assigned to the next Time Window(s). If two or more operations have the same estimated starting Time, the operation with a longer processing time will have a higher priority to be assigned to the current Time Window. If the processing time is equal, the operation with earlier order in its job will have a higher priority. If the order of the operations is the same, the operation that belongs to the smaller job number will have a higher priority. After decomposing the problem into sub-problems, we solved each sub-problem individually using a hybrid Answer Set Programming system (ASP [ref]), Clingo [DL], an extension of Clingo [ref] with different constraints. The proposed approach aimed to solve each sub-problem individually; to do that, we have applied a multi-shot ASP. We optimize the first Time Window and, after a predefined time out or reaching the optimum, proceed to the next Time Window, and so on until finishing all Time Windows. We performed experiments with different scale benchmark instances to test the performance of our proposed approach [ref taillard and swv]. The number of jobs and machines is between 30 – 100 and 10 – 20, respectively.

Table 1 shows the obtained results of our experiments, where the first column represents the number of Time Windows. The second column shows the comparison features; the first one represents the total completion time for all jobs (makespan), the second one illustrates the running Time in seconds, and the third one shows how many times the model didn’t reach the optimal solution per Time Window. Starting from columns 3 – 9, there are groups of instances with different sizes except columns 5 and 6, which have the same size. However, it should be split because it is noticed in the literature as Easy and Hard [ref]. Each group of the instances in the table has ten instances except for groups 50 X 10 (E) and (H) where each of them has five instances. The numbers in the table are the average of makespan, CPU time, and the number of interrupted calls. For example, group 30 X 15 with Time window (1), the values 2144.7, 1000.2, and 1 represent the average makespan, running Time, and the number of interrupted calls of all instances with 30 jobs and 15 machines, respectively. From Table 1, we can see that the values of makespan with one Time Window are the best in most cases. Increasing Time Windows provide solutions in a shorter time, but the quality of solutions is getting worse. However, when the problem size is increased, the values of makespan of more Time Windows are better than those with one Time Window. For example, the makespan value of the instances 100 X 20 with one and two Time Windows are 46786.1 and 23977.5, respectively. However, with three and four Time Windows are 7002.6 and 6964.4, respectively. In conclusion, decomposing the whole problem into sub-problems positively impacts, especially when the problem size is large.