The Job-Shop Scheduling Problem
Chapter 2 in *Empirical Modeling and Analysis of Local Search Algorithms for the Job-Shop Scheduling Problem* Ph.D. Dissertation

Jean-Paul Watson

Fall 2003

In the most general terms, a scheduling problem specifies a set of tasks to be performed using a finite set of resources. The objective is to schedule processing of all the tasks such that some overall measure of efficiency is maximized, e.g., the time required to complete all tasks. Scheduling is ubiquitous in real-world applications, ranging from operating systems to large-scale military logistics problems. The details of particular scheduling problems are highly variable, which led researchers to develop "distilled" scheduling problems, each abstractly representing a particular class of real-world problem. Although numerous abstract scheduling problems have been introduced (e.g., see the recent books by Błażewicz et al. [BSE$^+$96], Brucker [Bru01], and Pinedo [Pin01]), the most widely studied problem is the static, deterministic job-shop scheduling problem, hereafter referred to simply as the JSP. In the JSP there are n jobs and m machines. Each job must be processed on each machine exactly once, for a fixed duration, and in a pre-specified order; the processing of a job on a machine is called an operation. Each machine can only process a single operation at a time, and once initiated, an operation must be processed to completion. Various scheduling objectives have been introduced for the JSP. However, most research considers the objective of *makespan minimization*, i.e., minimizing the maximal completion time of all jobs [BDP96].

Although the JSP is the most widely studied abstract scheduling problem, it is also, paradoxically, one of the least realistic. Mattfeld [Mat96] notes the following restrictions inherent in the definition of the JSP, many of which need not hold in any given real-world scheduling problem (French [Fre82] provides a similar list):

- No two operations of one job may be processed simultaneously.
- No preemption (i.e., process interruption) of operations is allowed.
- No job is processed twice on the same machine.
- Each job must be processed to completion.
- Jobs may be started at any time, i.e., no release times exist.
- Jobs may be finished at any time, i.e., no due dates exist.
- Jobs must wait for the next machine in the processing order to become available.

- No machine may process more than one operation at a time.
- Machine setup times are negligible.
- There is only one type of each machine.
- Machines may be idle within the schedule period.
- Machines are available at any time.
- The machine processing orders of each job is known in advance and is immutable.

Consequently, algorithms for solving the JSP are of little direct use to practitioners, who must deal with the intricacies of their particular scheduling problem. McKay et al. [MSB88] even argue that "the [JSP] problem definition is so far removed from job-shop reality that perhaps a different name for the research should be considered." Yet, despite the lack of realism, the JSP has drawn intense attention from researchers in a variety of disciplines, ranging from operations research to computer science and mathematics; many of which are at best peripherally involved with real-world scheduling problems.

Why the intense attention on a problem of little practical significance? Mattfeld (p. 8, 1996) [Mat96] argues that "... benefit from previous research can only be obtained if a widely accepted standard model exists." However, the primary draw for most researchers is the fact that the JSP is notoriously difficult, widely accepted as empirically one of the most difficult NP-hard problems, and consequently poses a constant intellectual challenge: despite over 40 years of effort, resulting in hundreds of published journal articles and tens of dissertations, even state-of-the-art algorithms often fail to consistently locate optimal solutions to relatively small problem instances (e.g., see Jain [Jai98], Tables 2.4 through 2.10). The inherent intractability (unless $P = NP$) of the JSP is often illustrated by considering the history surrounding attempts to locate optimal solutions to a small benchmark problem instance introduced in 1963 by Fisher and Thompson [FT63], which was not solved to optimality until 1986. Jain and Meeran [JM99] provide an overview of the algorithmic developments leading to the solution of this famous problem instance.

In the remainder of this chapter, we introduce the job-shop scheduling problem and a number of related concepts that are integral to understanding and developing local search algorithms for its solution. We begin in Section 1 and introduce key sub-classes of problem instances in Section 2; empirical observations regarding the relative difficulty of JSP instances are discussed in Section 3. In Sections 4 through 7, we discuss the concepts, specification, and visualization of solutions and schedules, in addition to methods for quantifying the similarities between them. We conclude in Section 8 by discussing the notion of a critical path, a concept central to the design of local search algorithms for the JSP, which we discuss in Section **??**.

1 Definition, Notation, and Computational Complexity

In the $n \times m$ job-shop scheduling problem (JSP), there are n jobs, each of which must be processed exactly once on each of m machines. Each job i is routed through the m machines in some pre-defined order π_i, where $\pi_i(j)$ denotes the jth machine in the routing order; clearly, $1 \leq i \leq n$ and $1 \leq j \leq m$. Conversely, let $\pi_i^{-1}(j)$ denote the position of machine j in the routing order of job i. The processing of job i on machine $\pi_i(j)$ is denoted o_{ij} and is called an *operation*. An operation o_{ij} is processed on machine $\pi_i(j)$ for an integral duration $\tau_{ij} \geq 0$. For $2 \leq j \leq m$, o_{ij} cannot initiate processing until

	Operation					
Job	o_{i1}		o_{i2}		o_{i3}	
(i)	τ_{i1}	$\pi_i(1)$	τ_{i2}	$\pi_i(2)$	τ_{i3}	$\pi_i(3)$
1	5	1	8	2	2	3
2	7	3	3	1	9	2
3	1	1	7	3	10	2
4	2	2	3	3	1	1

Table 1: Specification of a 4×3 random JSP.

o_{ij-1} has completed. The constraints imposed by the job routing orders and unit capacity restrictions are respectively referred to as *precedence constraints* and *resource constraints*. In contexts where the job and routing order indices of an operation are irrelevant, we drop the dual subscript notation in favor of the single subscript notation, e.g., o_k or τ_k.

Each solution s to an instance Ω of the $n \times m$ JSP specifies a processing order for all of the jobs on each machine. There are $n!$ possible processing orders for each machine, resulting in $(n!)^m$ possible solutions to Ω. However, these solutions are often *infeasible* such that the combination of job routing orders π_i and the machine processing orders results in a cyclic ordering dependency between at least one pair of operations. Clearly, realizable job-shop schedules cannot be obtained from infeasible solutions. A solution s with no such cyclic dependencies is said to be *feasible*. We denote the set of all feasible solutions to an instance Ω by S_Ω.

Each solution $s \in S_\Omega$ implicitly specifies an earliest possible start time est_{ij} for each operation o_{ij} such that all precedence and resource constraints are satisfied; algorithms for computing the set of est_{ij} are discussed in Section **??**. The earliest possible completion time ect_{ij} for an operation o_{ij} is then given by $ect_{ij} = est_{ij} + \tau_{ij}$. The makespan $C_{max}(s)$ of a solution $s \in S_\Omega$ is the maximum earliest completion time of the last operation of any job i: i.e., $C_{max}(s) = max(ect_{1m}, ect_{2m}, \cdots, ect_{nm})$. The optimal makespan, denoted C^*_{max}, is equal to the minimal makespan of any solution in the set of all feasible solutions, i.e., $C^*_{max} = min_{\forall s \in S_\Omega}(C_{max}(s))$.

The decision problem of finding a solution to the JSP with a makespan less than or equal to some constant L is known to be NP-complete for $m \geq 2$ and $n \geq 3$ [GJS76]. Consequently, the optimization problem of locating a solution with a makespan equal to C^*_{max} is (strongly) NP-hard. Even accurate approximation in the JSP is provably intractable: unless $\mathcal{P} = \mathcal{NP}$, no polynomial-time algorithm exists that can guarantee a solution to the JSP within 20% of the optimal makespan [WHH$^+$97].

2 Specification and Classification of Problem Instances

An instance Ω of the $n \times m$ JSP is completely specified by the set of nm operation durations τ_{ij} and n job routing orders π_i. We show the specification for a small 4×3 instance originally introduced by Foo and Takefuji [FT88] in Table 1. Beginning with Fisher and Thompson [FT63], researchers typically generate problem instances by sampling the τ_{ij} independently and uniformly from a fixed-width interval $[LB, UB]$. Most often $LB = 1$ and $UB = 99$, e.g., see Taillard [Tai93] or Demirkol et al. [DMU98]. The definition of the JSP places no *a priori* restrictions on the form of the job routing orders

π_i. Consequently, researchers frequently construct the π_i by generating random permutations of the integers $[1..m]$. We define *random* JSPs as instances constructed by (1) sampling the τ_{ij} uniformly from a fixed-width interval and (2) generating the π_i from random permutations of the integers $[1..m]$. Most well-known JSP benchmark instances are random JSPs. The instance shown in Table 1 is a random JSP, with the τ_{ij} sampled from the interval $[1, 10]$.

Researchers have introduced specializations of the random JSP by imposing specific constraints on the job routing orders that more accurately reflect features found in some real-world job shops. In these more structured JSPs, the set of m machines is partitioned into *wf* contiguous, equally-sized subsets called *workflow partitions*. Clearly, $wf = 1$ in random JSPs. In contrast, when $wf = 2$, the set of m machines is partitioned into two subsets containing the machines 1 through $m/2$ and $m/2 + 1$ through m, respectively. In these problems, every job must be processed on all machines in the first partition before proceeding to any machine in the second partition. No constraints are placed on the job routing orders within each partition. We refer to JSPs with $wf = 2$ simply as *workflow* JSPs.

In a *flowshop* JSP, $wf = m$, and each job must visit the machines in an identical pre-determined order. The flowshop JSP is closely related to the Permutation Flow-Shop Problem (PFSP), in which the job processing orders for all machines is identical. However, the optimal makespan to a PFSP instance is typically higher than the optimal makespan of the corresponding flowshop JSP, where the job processing orders of different machines can be variable. For most benchmark problems, the difference is relatively small. The worst-case difference is substantially larger; Potts et al. [PSW91] show that the optimal makespan of a PFSP can exceed that of the corresponding flowshop JSP by a factor of more than $\frac{1}{2}\sqrt{m}$.

To date, no research has been directed toward generating problem instances with structured τ_{ij}, although it is well-known that real-world scheduling problems typically possess non-random operation processing times [PDS73] [DPS92]. Further, there is evidence that problem structure can have a significant impact on the algorithm performance [WBWH02]. Although we previously introduced a method for generating structured permutation flow-shop scheduling problems [WBWH02], which can be easily extended to the JSP, we do not consider such problem instances here. The present goal is to analyze and understand the large body of research already performed on the JSP, which necessarily constrains analysis to existing classes of problem instances.

3 The Relative Difficulty of Problem Instances

While developing algorithms for solving the JSP, researchers have made numerous qualitative observations regarding the relative difficulty of different problem instances. With few exceptions, these observations appear to be algorithm-independent, such that relative difficulty with respect to constructive algorithms such as branch-and-bound also holds for local search. For our purposes, these observations serve to validate the models of problem difficulty for local search that we develop in Chapters **??** through **??**. In particular, our models should at *least* be consistent with these observations, and ideally should provide *explanations* (relative to local search algorithms) for the observed behavior.

The primary observations regarding the relative difficulty of JSP instances, which have emerged from nearly 50 years of research, are as follows:

- Given fixed n and m, workflow JSPs are typically more difficult than random JSPs.
- Given fixed n and m, flowshop JSPs are typically more difficult than workflow JSPs.

- Given fixed m and *wf*, *square* (i.e., $n/m \approx 1$) JSPs are typically more difficult than *rectangular* (i.e., $n/m \gg 1$) JSPs.
- Given fixed n, m, and *wf*, relative problem difficulty is largely algorithm-independent.

We now describe each of these observations in more detail, noting in advance that although researchers have proposed intuitive explanations for each of these observations, rigorous causal explanations remain elusive.

Workflow and flowshop JSPs differ from random JSPs in that the job routing orders π_i are more structured. While the presence of structure often makes problems *easier* to solve, assuming the existence of an algorithm that exploits this structure, this is not the case in the JSP. Specifically, given fixed, arbitrary n and m, the average difficulty of problem instances, as measured either by the cost required to locate an optimal solution or to *prove* optimality of a solution, is proportional to *wf*, i.e., those instances that possess more workflow partitions are likely, on average, to be more difficult. In particular, the most difficult instances are flowshop JSPs, where $wf = m$. Evidence for this observation stems from a wide variety of sources. For example, Storer et al. [SWV92] introduced sets of 50×10 random and workflow JSPs in 1992; since then, the random instances have been solved to optimality, while the optimal makespans of all but one of the workflow instances are unknown. Similarly, the most difficult 10×10 benchmark instances, Fisher and Thompson's infamous 10×10 instance and the `orb` instances introduced by Applegate and Cook [AC91], are all "nearly" workflow or flowshop instances, in that the requirement that a job be processed on all machines in one workflow partition before proceeding to any machine in the next workflow partition is occasionally violated.

Large differences in the relative difficulty of square (i.e., $n/m \approx 1$) versus rectangular (i.e., $n/m > 1$) JSPs are easily illustrated by considering the best-known makespans for the problems in Taillard's benchmark suite of JSP instances, which we discuss in Section **??**. Here, the optimal makespans of all the relatively small 20×20 and 30×20 instances are currently unknown, while the optimal makespan for all of the larger 50×15, 50×20, and 100×20 instances are known, despite astronomical differences in the sizes of the search space. Taillard [Tai94] performed an in-depth analysis of changes in relative difficulty as n and m are varied, concluding that for a tabu search algorithm, the cost of locating optimal solutions grows *polynomially* with increases in n and m when $n/m \geq 6$. In contrast, when $n/m \approx 1$, increases in n (or equivalently m) yield apparently exponential increases in the cost required by the same tabu search algorithm to locate optimal solutions.

The third observation stems from the fact that in a set of fixed-sized problem instances with a given *wf*, the easy (difficult) instances are easy (difficult) for *all* search algorithms. So-called "open" problem instances, i.e., those for which the optimal makespan is not known, have resisted solution by numerous algorithms based on dramatically different computational paradigms. Similarly, when the optimal makespan is known, the computational effort required to locate an optimal solution is consistently predictable from the historical reputation of the difficulty of a given instance.

The class of JSP instances that are considered "intractable", in that the optimal makespan has not been determined, clearly depends on the state-of-the-art in available computing power. Jain and Meeran [JM99] indicate that a problem instance can be considered hard or intractable if it possesses the following properties:

- $n \cdot m \geq 200$
- $n \geq 15$

- $m \geq 10$
- $n < 2.5m$

They further observe that "The problem is made more intractable when [*wf*=2] and in such a situation n need not be less than or equal to $2.5m$" ([JM99], p. 407). Although this classification was introduced in 1998, it is still valid: significant advances in computing power has *not* significantly expanded the range of tractable JSP instances. It is currently beyond the power of exact algorithms to consistently determine optimal makespans to 20×15, 20×20, and 30×15 benchmark instances, and approximation algorithms have great difficulty locating optimal or even high-quality sub-optimal solutions to these same instances. Even though smaller instances, e.g., 15×15 are now considered tractable, this does *not* necessarily imply that it is easy for approximation algorithms to locate optimal solutions; rather, it is possible for *some* approximation algorithms, likely after extended periods of computation, to *occasionally* locate an optimal solution.

4 Solutions: Specification, Properties, and Notation

One approach to specifying solutions to an $n \times m$ instance Ω of the JSP is to simply list the job processing orders γ_j for each machine j, $1 \leq j \leq m$. For example, an optimal solution to the 4×3 instance shown in Table 1 is given as follows:

- $\gamma_1 = (1, 3, 2, 4)$
- $\gamma_2 = (4, 1, 2, 3)$
- $\gamma_3 = (2, 3, 4, 1)$

Each machine processing order γ_j is simply an ordered permutation of the integers $[1..n]$ such that $\gamma_j(i)$ denotes the ith job in the processing order of machine j. Conversely, let $\gamma_j^{-1}(i)$ denote the position of job i in the processing order γ_j of machine j.

The precedence constraints imposed by the specification of the instance Ω introduce at most one predecessor and one successor operation for any operation o_{ij}. We denote the corresponding "job" predecessors and successors of o_{ij} respectively by JP_{ij} and JS_{ij}, which are defined as follows:

- $\forall i \neq 1, JP_{ij} = o_{i(j-1)}$
- $\forall i \neq m, JS_{ij} = o_{i(j+1)}$
- $JP_{i1} = JS_{im} = *$

where $*$ denotes a "null" or non-existent operation.

Similarly, for any operation o_{ij}, the set of machine processing orders γ_j introduce at most one additional predecessor and successor operation apiece. We denote the corresponding "machine" predecessors and successors of o_{ij} respectively by MP_{ij} and MS_{ij}, which are defined as follows, given that o_{ij} is processed on machine $x = \pi_i(j)$ at position $y = \gamma_x^{-1}(i)$:

- $\forall y \neq 1, MP_{ij} = o_{\gamma_x(y-1)\pi_{y-1}^{-1}(x)}$

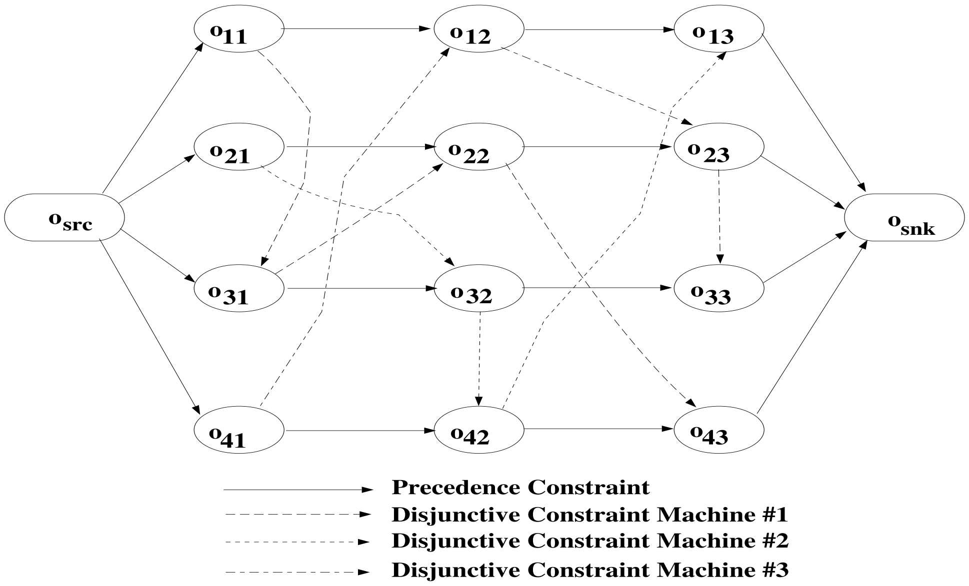

Figure 1: The disjunctive graph for an optimal solution to the 4×3 JSP instance shown in Table 1.

- $\forall y \neq n, MS_{ij} = o_{\gamma_x(y+1)\pi^{-1}_{y+1}(x)}$
- if $y = 1$, $MP_{ij} = *$
- if $y = n$, $MS_{ij} = *$

where $*$ again denotes a non-existent operation.

An alternative method for specifying solutions is via a *disjunctive graph*. Instead of explicitly specifying a job processing sequence for each machine, a disjunctive graph specifies the relative processing order for each *pair* of jobs on a machine. Further, a disjunctive graph (redundantly) specifies the precedence constraints imposed by the job processing orders π_j. Formally, a disjunctive graph is a directed graph $G = (V, C \cup D)$. The vertex set V contains $nm + 2$ vertices, where nm of these vertices represent the operations of the n jobs, while the remaining two vertices represent dummy operations known as the *source* and the *sink*, which we respectively denote o_{src} and o_{snk}. Both o_{src} and o_{snk} are orphaned, in the sense that they do not belong to any job and are not processed on any machine, such that $\tau_{src} = \tau_{snk} = 0$. A precedence constraint is defined between (1) o_{src} and each operation o_{1j} and (2) each operation o_{im} and o_{snk}.

The edge set C contains a directed edge for each precedence constraint imposed by the job routing orders and the o_{src} and o_{snk} operations. The edge set D contains a directed edge between each distinct pair of operations that are processed on the same machine. The edges $(o_{ij}, o_{ik}) \in C$ are oriented such that $j < k$. Edges in C imposed by the source and sink operations are respectively oriented as (o_{src}, o_{1j}) and (o_{im}, o_{snk}). The edges $(o_{ij}, o_{kl}) \in D$, by definition processed on the same machine $x = \pi_i(j) = \pi_k(l)$, are oriented such that $\gamma_x^{-1}(i) < \gamma_x^{-1}(k)$. In Figure 1, we show a disjunctive graph for an optimal solution, identical to the example solution specified via $\gamma_1 \cdots \gamma_3$ above, to the instance specified in Table 1.

The job processing order and disjunctive graph specifications are equivalent, in that one can be easily transformed into the other. Both methods can also be used to *represent* solutions in a search algorithm. For example, the source and sink operations serve to equip the disjunctive graph with a single entry and exit point, which in turn facilitates computation of various solutions attributes, including feasibility and operation start times. Although neither method is *directly* used to represent solutions for the local search algorithms we consider, aspects of both methods are incorporated, e.g., see Section **??**. Further, the methods are used to compare and contrast various solutions, as discussed below in Section 5.

5 Quantifying Solution Similarity

The disjunctive graph specification of a solution also provides a straightforward measure of the similarity or "distance" between two solutions. Recall that the precedence constraints are independent of any particular solution to a given problem instance. Thus, two solutions differ only in the orientation of their disjunctive (directed) edges. For any solution $s \in S_\Omega$, the orientation of a disjunctive edge between two jobs i and j on machine k can be represented as a Boolean variable, which we denote $\mathit{precedes}_{i,j,k}(s)$. If job i is processed before job j on machine k in solution s, $\mathit{precedes}_{i,j,k}(s) = \mathit{true}$; otherwise, it is false. The distance $D(s_1, s_2)$ between two solutions $s_1, s_2 \in S_\Omega$ to an $n \times m$ JSP Ω is then given by

$$\sum_{i=1}^{m} \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \mathit{precedes}_{i,j,k}(s_1) \oplus \mathit{precedes}_{i,j,k}(s_2) \tag{1}$$

where the symbol $\oplus$ denotes the Boolean XOR operator. We denote the normalized distance $\frac{2 \cdot D(s_1,s_2)}{n(n-1)}$ by $\overline{D}(s_1, s_2)$ ($0 \leq \overline{D}(s_1, s_2) \leq 1$). Equation 1 can also be used to compute the distance between pairs of infeasible solutions or between infeasible and feasible solutions. Finally, nothing in the definition of Equation 1 is specific to the disjunctive graph – the distance can also be computed when solutions are specified as machine permutations; the term "disjunctive graph distance" is used primarily for historical reasons.

6 Solutions versus Schedules and Schedule Taxonomy

A feasible *solution* $s \in S_\Omega$ specifies a job processing order for each machine in which there exist no cyclic ordering dependencies between pairs of operations. Any reference to time in a solution is implicit – an earliest possible start time can be computed for each operation. In contrast, a (feasible) *schedule* specifies a start time for each operation of each job such that the precedence and resource constraints are satisfied. In general, there is a one-to-many relationship between solutions and schedules, as there is no requirement to start operations at the earliest possible time; assuming an infinite time horizon, there are infinitely many possible schedules arising from any given solution. We refer to schedules in which operations are started later than their earliest possible start time as *inadmissible*. However, to achieve the makespan $C_{max}(s)$ it is necessary to schedule some subset of the operations at their earliest possible start time; the exact subset is introduced in Section 8. Consequently, operations are in practice assigned to their earliest possible start time; the resulting schedules are known as admissible or *semi-active*. Clearly, there exists a one-to-one relationship between solutions and (admissible) semi-active schedules; we use the two terms interchangeably.

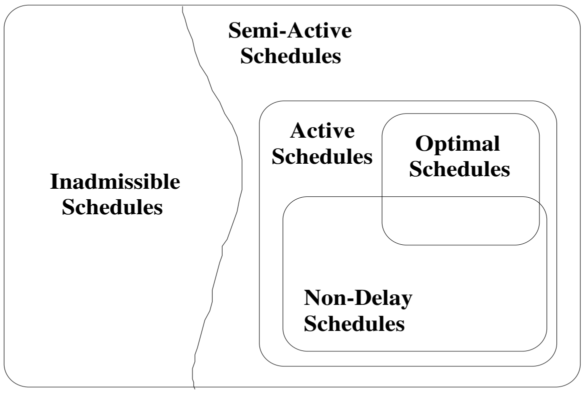

Figure 2: The hierarchy of schedules in the JSP.

In a semi-active schedule, the only way to reduce the makespan is to re-order the job processing order on at least one machine. In particular, it may be possible to move an operation earlier than its assigned start time *without* delaying the start time of any other operation, e.g., if there is idle time in the middle of a machine sequence. Such a move is called a *global left shift*, and a schedule in which no global left shifts are possible is called an *active* schedule. A schedule in which no machine is kept idle if there is an operation available for processing is called a *non-delay* schedule. By definition, all non-delay schedules are also active schedules.

We show the taxonomy of schedules in the JSP in Figure 2. At least one optimal schedule is necessarily active [GT60], but there may exist no optimal schedule that is non-delay; Fang [Fan94] provides counter-examples. Although restricting search to the set of active solutions appears advantageous, it is not always practical; the problem of verifying that a solution is active, i.e., checking that no global left-shifts are possible, is NP-complete [Vae95]. Consequently, most local search algorithms for the JSP restrict search to the space of semi-active solutions. Nevertheless, active and non-delay schedules do play an important role in local search, as we discuss in Section **??**.

7 Visualizing Schedules

Solutions can be easily specified via a set of job processing orders or a disjunctive graph; however, in both cases the notion of time is only implicit. Although we could easily annotate these specifications with the assigned start time in the corresponding schedule (again, we assume semi-active scheduling), the result is difficult to interpret and fails to provide immediate insight into the qualitative nature of the overall schedule. To more effectively visualize schedules, we turn to a scheme known as a Gantt chart [Gan19], which simply records the activity of each machine over the duration of the schedule. In Figure 3, we provide a Gantt chart visualization of the earliest start-time schedule for an optimal solution, identical to that shown in Figure 1, to the 4×3 instance specified in Table 1; here, $C^*_{max} = 32$. The activity of each machine is indicated on a horizontal "time-line", where gray and white areas respectively represent active and idle times. Gantt charts can be used to identify bottleneck machines, potential flexibility in the schedule, and critical paths, as we discuss next in Section 8.

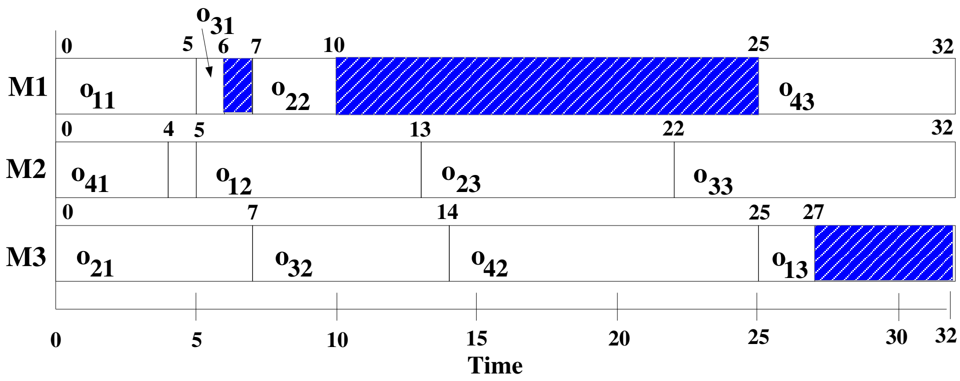

Figure 3: The Gantt chart visualization of the earliest start-time schedule for an optimal solution to the 4×3 JSP instance specified in Table 1. Shaded regions represent machine idle time.

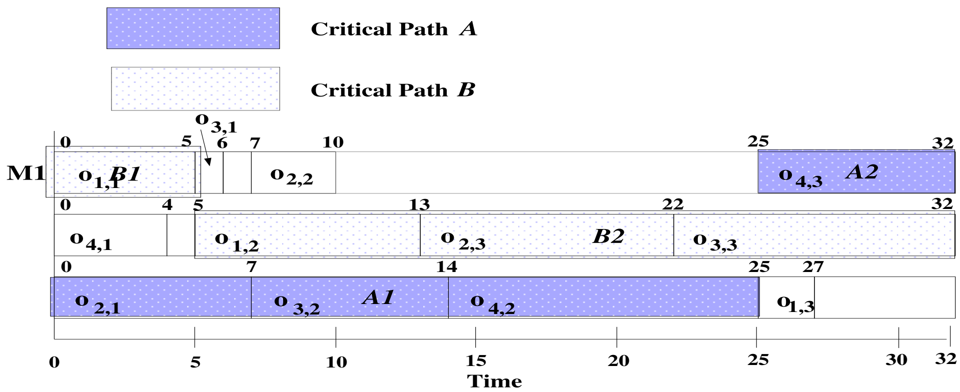

Figure 4: An earliest-start-time schedule for an optimal solution to the JSP instance shown in Table 1. The solution has two critical paths, labeled **A** and **B**.

8 Critical Paths and Critical Blocks

A semi-active schedule is generated by starting operations in a solution s at their earliest possible time. However, it is possible to start some subset of operations later, yet still yield a schedule with makespan $C_{max}(s)$. Let lft_{ij} denote the *latest finish time* of operation o_{ij} such that if all operations o_{ij} complete processing by lft_{ij} the resulting solution will have a makespan equal to $C_{max}(s)$; as with the earliest start times, the latest finish times are implicitly specified by s. Define the *latest start time* lst_{ij} of an operation o_{ij} as $lft_{ij} - \tau_{ij}$. In order to generate a schedule from s with makespan $C_{max}(s)$, all of the o_{ij} must be assigned a start time in the interval $[est_{ij}, lst_{ij}]$ such that the selected times are mutually consistent, i.e., all precedence and disjunctive constraints are satisfied.

The set of operations o_{ij} such that $est_{ij} = lst_{ij}$ are known as *critical* operations. If the actual start time of a critical operation is delayed by even a single time unit, then the makespan of the resulting schedule is guaranteed to be larger than $C_{max}(s)$. A contiguous sequence of critical operations starting at time $t = 0$ and ending at time $t = C_{max}(s)$ is called a *critical path*, such that no machine idle time exists along the path. In general, a solution s can have multiple critical paths, and a single critical operation can be shared by multiple critical paths.

Alternatively, we can define the *time-to-end* tte_{ij} of an operation o_{ij} in s as the difference between $C_{max}(s)$ and the latest possible end time of o_{ij} such that the schedule s is still feasible and has a makespan not exceeding $C_{max}(s)$. Consider a solution s and the *inverse* s' of s, obtained by inverting all of the edges in the disjunctive graph of s, such that the roles of o_{src} and o_{snk} are exchanged. The tte_{ij} of s are then equivalent to the est_{ij} of s'. Under this notation, an operation is critical if and only if $est_{ij} + \tau_{ij} + tte_{ij}$, i.e., when there is no "slack" in the scheduling of o_{ij}.

In Figure 4, we show the earliest-start-time schedule for an optimal solution (identical to that shown in Figure 3) to the JSP instance shown in Table 1. The solution has two critical paths, **A** and **B**. Each critical path is composed of sub-sequences of critical operations on the same machine; these sub-sequences are called *critical blocks*. Critical path **A** is composed of critical blocks on machines 3 and 1, containing three operations and a single operation, respectively. Analogous, critical path **B** is composed of critical blocks on machines 1 and 3, respectively containing one and three operations. No critical operation is shared by more than one critical path.

Critical paths play a central role in most high-performance algorithms for the JSP, for the following reason: given a complete and feasible solution $s \in S_\Omega$, the *only* way to reduce $C_{max}(s)$ is to re-order the sequence of operations on all critical paths [vLAL92]. Re-ordering the operations along a *single* critical path is *not* guaranteed to reduce $C_{max}(s)$, although several researchers implicitly make this assumption. The latter only holds when some of the critical operations on the path under consideration are shared by all critical paths; by re-ordering the operations on a random critical path, we are implicitly re-ordering the operations on all critical paths.

References

[AC91] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[BDP96] J. Błażewicz, W. Domschke, and E. Pesch. The job shop scheduling problem: Conventional and new solution techniques. *European Journal of Operational Research*, 93:1–33, 1996.

[Bru01] P. Brucker. *Scheduling Algorithms*. Springer Verlag, 3rd edition, 2001.

[BSE+96] J. Blazewicz, G. Schmidt, K.H. Ecker, E. Pesch, and J. Weglarz. *Scheduling Computer and Manufacturing Processes*. Springer Verlag, 1996.

[DMU98] E. Demirkol, S.V. Mehta, and R. Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109:137–141, 1998.

[DPS92] R.A. Dudek, S.S. Panwalker, and M.L. Smith. The lessons of flowshop scheduling research. *Operations Research*, 40(1):7–13, 1992.

[Fan94] H. Fang. *Genetic Algorithms in Timetabling and Scheduling*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[Fre82] S. French. *Sequencing and Scheduling – An Introduction to the Mathematics of the Job-Shop*. Ellis Horwood, John-Wiley & Sons, 1982.

[FT63] H. Fisher and G.L. Thompson. *Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules*, chapter 15, pages 225–251. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

[FT88] S. Y. Foo and Y. Takefuji. Stochastic neural networks for solving job-shop scheduling: Part 1. problem representation. In Bart Kosko, editor, *IEEE International Conference on Neural Networks*, volume 2, pages 275–282, 1988.

[Gan19] H. L. Gantt. Efficiency and democracy. In *Transactions of the American Society of Mechanical Engineers*, volume 40, pages 799–808, 1919.

[GJS76] M.R. Garey, D.S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[GT60] B. Giffler and G. L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8:487–503, 1960.

[Jai98] A.S. Jain. *A Multi-Level Hybrid Framework for the Deterministic Job-Shop Scheduling Problem*. PhD thesis, University of Dundee, 1998.

[JM99] A.S. Jain and S. Meeran. Deterministic job-shop scheduling: Past, present, and future. *European Journal of Operational Research*, 113:390–434, 1999.

[Mat96] D.C. Mattfeld. *Evolutionary Search and the Job Shop*. Physica-Verlag, Heidelberg, 1996.

[MSB88] K.N. McKay, F.R. Safayeni, and J.A. Buzacott. Job-shop scheduling theory: What is relevant? *Interfaces*, 18(4):84–90, 1988.

[PDS73] S.S. Panwalker, R.A. Dudek, and M.L. Smith. Sequencing research and the industrial scheduling problem. In *Proceedings of Symposium on Theory of Scheduling and its Applications*, pages 29–38. Springer-Verlag, New York, 1973.

[Pin01] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, 2nd edition, 2001.

[PSW91] C.N. Potts, D.B. Shmoys, and D.P. Williamson. Permutation vs. non-permutation flow shop schedules. *Operations Research Letters*, 10:281–284, 1991.

[SWV92] R. H. Storer, S. D. Wu, and R. Vaccari. New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, 38:1495–1509, 1992.

[Tai93] E.D. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993.

[Tai94] E.D. Taillard. Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing*, 6(2):108–117, 1994.

[Vae95] R.J.M. Vaessens. *Generalized Job Shop Scheduling: Complexity and Local Search*. PhD thesis, Eindhoven University of Technology, 1995.

[vLAL92] P.J.M van Laarhoven, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40(1):113–125, 1992.

[WBWH02] J.P. Watson, L. Barbulescu, L.D. Whitley, and A.E. Howe. Contrasting structured and random permutation flow-shop scheduling problems: Search space topology and algorithm performance. *INFORMS Journal on Computing*, 14(2):98–123, 2002.

[WHH$^+$97] D.P. Williamson, L.A. Hall, J.A. Hoogeveen, C.A.J. Hurkens, J.K. Lenstra, S.V. Sevast'yanov, and D.B. Shmoys. Short shop schedules. *Operations Research*, 45:288–294, 1997.