LICENTIATE THESIS

A Decision Support System for Scheduling a Shop Floor Work Center

MAGNUS OSKARSSON

Department of Mathematics
CHALMERS UNIVERSITY OF TECHNOLOGY
GÖTEBORG UNIVERSITY

January 1999
NO 1999-4/ISSN 0347-2809

Abstract

We suggest a scheduling application that is to be used as a decision support system in a shop floor work center, mainly in case of a production disturbance. In a decision support situation, a small set of scheduling problems are identified corresponding to different global actions. Each of these problems are fed to a solver and the overall best solution found with respect to total costs is proposed to the user after a short execution time. The total cost includes activity costs, penalty costs for late product deliveries, and costs related to global actions. The scheduling problems have to be modelled on a detailed level, including features like multiple resource usage for operations (e.g. machine, operator and fixture) with possibly different time usage for the resources involved in the operation, volumetric resources, limited buffers, flexible routing, product lot sizes, release dates for individual products and delayed arrival of resources.

We give an overview of different existing techniques for modelling and scheduling of manufacturing processes, and discuss their potential of being extended to meet the modelling and solution demands of our application. Based on these discussions, we choose a Petri net for the modelling of the scheduling system. In order to be able to include all the desired modelling features, we introduce a new type of timing mechanism in the net, where time is associated to individual tokens and output arcs rather than with places or transitions as in standard timed Petri nets. Several search algorithms suitable for a limited execution time are presented, and their performance is evaluated on a set of test problems with respect to different objective functions and evaluation functions. A solution method based on beam search produces the overall best results. For the objective of minimizing tardiness penalties, an approximate evaluation function is presented that together with beam search gives very good results. A heuristic based on a limited lookahead in time is used in all algorithms, and the results show that a short but non-zero lookahead is favourable in most situations.

Keywords and phrases: Production scheduling, Decision support system, Combinatorial optimization, Petri nets, Heuristic search.

AMS 1991 subject classification: 90B35, 90B50, 90C27.

Preface

This report

This report summarizes the results from the first phase of a research project which is a joint venture between the Departments of Mathematics and Computing Science at Chalmers University of Technology and Göteborg University, and Prosolvia Systems AB, Göteborg, Sweden. The university part of the project has to a large extent been financed by NUTEK, the Swedish National Board for Industrial and Technical Development.

The report also serves as the thesis of the ECMI[1] post-graduate program in Industrial Mathematics at Chalmers, fulfilling the requirements for the degree of Licentiate of Engineering. This five-semester program includes a block of core courses covering several areas of applied mathematics and computing science and a block of specialization courses within one selected field, which in this case is algorithms and combinatorial optimization. The final and main part of the program is to work with a problem from industry.

The project

The long-term goal of this research project is a new kind of product for scheduling of manufacturing systems. The core of the product is a scheduling problem solver, around which a decision support logic is built. The aim of the first phase of the project has been to lay the theoretical foundation for the future research and development of the product, with emphasis on the solver. More specifically, the work has been focused on

- specifying the requirements of the product
- studying modelling and solution methods for similar problems
- suggesting a model for the system
- developing solution methods for the scheduling problems and evaluating the results from tests
- identifying areas in need of future research

[1]The European Consortium for Mathematics in Industry

Outline of the paper

In Chapter 1, an introduction to the problem is given. Some graphical modelling techniques that can be used for describing production processes are presented in Chapter 2. Scheduling theory and commonly studied problems are described in Chapter 3. Models and solution methods for scheduling that have been studied in the literature are reviewed in Chapter 4. A reader well familiar with scheduling techniques in general and Petri net based scheduling in particular may skip most sections in the last three mentioned chapters without too much trouble, if it is desired to come to the new contributions of this paper as soon as possible. In Chapter 5, a model for the scheduling problems is presented, and its use within the decision support system is briefly discussed. A number of candidate solution methods for the scheduling problems are described in Chapter 6, and their performance on a set of test problems is discussed in Chapter 7. In Chapter 8, conclusions are made and suggestions for future research are presented in Chapter 9.

Acknowledgements

First of all, I would like to thank my supervisors for this project, Dag Wedelin at the Department of Computing Science and Roland Bruks at Prosolvia, for many fruitful discussions and ideas. I would also like to thank Bo Johansson at the Department of Mathematics for initiating and supporting the project, Torbjörn Liljenvall at the Department of Signals and Systems for introducing me to his work and giving me some good Petri net and scheduling literature, my ECMI colleagues for being such good work-mates, and finally my wife Annika for all the love and support.

Göteborg, January 1999

Magnus Oskarsson

Contents

Chapter 1

Introduction

In this chapter, we will first give an introduction to the problem area of *planning* and *scheduling*. Then we will give some background to the problem treated in this report, before describing it in more detail. Finally, we will informally present a possible solution model in order to give a picture of the character of the problem.

1.1 Planning and Scheduling

In the next section, we will describe some planning and scheduling problems encountered in the manufacturing industry. Although the reader will probably already have a good picture of what the terms planning and scheduling mean, they still deserve some discussion. The word planning is often used as a very broad term, including among other things the problem of scheduling. But it is also common to make a slight distinction between them, saying that planning is mainly concerned with "what to do", whereas scheduling concentrates on "how to do it". Following this "definition", we can say that a planning problem is on a more general level and its solution is often the input to one or more scheduling problems that take into account more details.

In this paper, the main focus will be on scheduling. We will in later present scheduling terminology and theory in more detail, but we will already here give some definitions on a very general level. Baptiste and Le Pape (1995) state that "scheduling is the process of assigning *activities* to *resources* in time". In our field of application, resources represent machines, tools, robots, operators, buffers etc., and activities are e.g. the processing of products in machines or the transport of products between machines. In a scheduling problem we often want to optimize some criterion while obeying a set of constraints. Again following Baptiste and Le Pape (1995), we can identify three main families of problems:

- In *(pure) scheduling problems*, the resources that are to be used for activities are uniquely specified together with their available capacity over time. The problem is to position the activities in time without exceeding resource capacity or violating any other constraints. This problem type is very common in manufacturing.
- In *(pure) resource allocation problems* the demand for different resource types over time is known and the problem is to allocate individual resources in time so that the supply equals or exceeds the demand and other constraints are

obeyed. This problem type is not very interesting in our case, but there are many important applications from other fields, e.g. airline crew scheduling.

- In *combined scheduling and resource allocation problems* it is to some extent possible to choose which activities that are to be done and/or decide which resources to allocate for activities. In manufacturing with *flexible routing* for products we have this kind of problem. From now on, we will mean this case when we talk about scheduling, although many examples will be pure scheduling problems.

1.2 Problem background

Planning and scheduling problems in the manufacturing industry can roughly be grouped into three levels, with respect to the scale of the model and the length of the time horizon involved:

The strategic level. At this level, typical problems are:

- What will the demand be for the company's products?
- Which factories should manufacture a certain product, and if they do, at what quantities?
- How should transports of raw materials and products be handled?
- If a new factory is needed to increase capacity, where should it be located?

The time horizon at this level is typically years. There is a large portion of uncertainty in the data. Except for perhaps the logistics oriented problems, most of the problems are very specific for a business area or even an individual company. This means that there are not many generally applicable tools available for solving these kinds of problems.

The factory (shop) level. Given the restrictions imposed by a strategic plan, the problems on this level are concerned with how to make the production as efficient as possible without getting delayed deliveries for customers. The time horizon is usually weeks or months. A planning task is to decide what products to produce in different time periods and how much, given customer orders, stock levels etc. The outcome of this is a number of *jobs*[1] that have to be performed, the times when the necessary raw materials and components for these jobs must be available and the times when it is desired that the jobs are finished. This data together with information on the availability of production resources forms the basis of a scheduling problem. Due to its large size this problem is usually not broken down into the finest details, instead a number of simplifications are done to make the problem manageable. The degree of simplification depends on the solution method used: simple rule-based methods allow more detail, whereas more advanced optimization methods require a simpler model. Due to variations in processing times, problems not predicted by the simplified model, unforeseen events etc., a schedule does not usually last very long and it has to be adjusted quite often. Methods taking some elements of uncertainty into account can produce more robust schedules. These kinds of

[1]a job corresponds in most cases to the manufacturing of a product.

problems can be quite similar even for companies producing entirely different goods, so a lot of more or less general scheduling strategies and methods have been developed over the years. Consequently, a large number of commercial computer-based tools for this are available, and a lot of research is performed in the area.

The work center level. The factory is usually divided into smaller parts, which we will refer to as *work centers*. They typically include a number of workers (we will call them *operators*), a few machines, and some transport devices and storage facilities for products. Other things such as robots, fixtures and tools might also be vital for the functionality of the work center. The scheduling problem on this level has similar prerequisites as the one at the factory level: the jobs to be performed, arrival times and desired completion times of these jobs, and the availability of resources. The time horizon is hours or days. But at this level, the foreman of the center is responsible for actually performing the jobs down at the shop floor. In many cases this is not very difficult since enough time is available from the overlying factory schedule. Also, the degree of uncertainty involved in the problem is not very high due to the short time horizon. For these cases, no computer-based solution methods are needed.

However, if an unexpected event actually does occur (a raw material gets delayed, a machine breaks down, an operator gets ill etc.) it might not be easy to follow the factory schedule, if possible at all. It can also be the case that the simplifications made in the factory level scheduling cause problems. In these cases scheduling on the work center level becomes a critical problem, since costly production disturbances on the factory level are threatening. At the work center level, as many problem details as possible must be taken into account in the scheduling model in order to make it useful. It might also be necessary for the foreman to take special actions in order to avoid delays. Another complicating factor is that there is not much time available for doing the scheduling. This is the problem of our attention, and it has (to the best of our knowledge) not been widely studied before. A good scheduling method for this kind of detailed but not necessarily very large problems can be useful also in other cases, e.g. if we have a situation where the production process of the work center is very complex, so that schedule optimization can increase productivity a lot.

We note here that some scheduling problems on the work center level have been studied earlier, especially in the case of Flexible Manufacturing Systems (FMS)[2]. We will review several papers on this subject in Chapter 4. Even if we are more interested in manned systems, there are many similarities with FMS scheduling. However, there are some limitations of these models, such as that most of them use scheduling objectives that are not relevant to us. Also, the possibilities of special events and actions are not part of these models, so the application we will describe lies within a to a large extent unexplored research field.

We also note that the idea of being able to respond to unexpected events is not new. On the contrary, there is a quite wide-spread approach to these problems called *reactive scheduling* (see e.g. Smith (1995)) that includes methods for adjusting

[2]FMS is an approach to manufacturing which allows production to be switched rapidly from one product to another. The system is usually fully automated, and consists of a few machines with changeable tools and some transport device (typically a robot).

existing schedules as the environment changes. However, these systems work on the factory level, both with respect to detail resolution and time horizon.

1.3 Problem description

We will now describe a decision support system for scheduling a work center, a system that is mainly intended to be used in case of a production disturbance. Its typical use will be that the user consults the decision support system when an unexpected event occurs. If not registered through an automatic alarm or the like, the user feeds the information regarding the event into the system. The system then identifies a few *scenarios* corresponding to special actions[3] relevant to the event. For each of these scenarios, a scheduling problem is solved with the objective of minimizing (the sum of) activity costs and costs for late deliveries. The cost of the best schedule found for the scenario is added to the costs related to the special action, if any. Finally, after a few seconds of total computing time it presents the scenario with the lowest total cost as the suggestion to the user. It should also be possible to use the system even if no special event has occurred, in which case there will be only the standard scenario. The time horizon considered is intended to be fairly short, sometimes only the remainder of the day.

A solution that appears good on the local level might be very bad in a global perspective. Although this system works locally, the scheduling objective used is focused on making decisions that are good also globally. The desired interaction with other parts of the factory can be achieved by setting appropriate delivery times for the products and associated penalty costs for late deliveries.

1.3.1 Why decision support?

Why is a (fast) decision support system like this needed? We see several motives:

- In case of a production disturbance, there can be huge extra costs for the company if actions are taken too late, or if bad decisions are made. There is thus a large money-saving potential for a decision support system that works well.

- Estimating the consequences of different actions is a very complex problem, even for an expert on production. In most cases it will be the foreman of the work center that makes the early important decisions, a task for which he probably has no training for. Also, the foreman might not have all information relevant for the decision-making.

- The difficulty of the problem is not alleviated by the fact that the decision for which actions to take must be made very quickly. This means that the decisions have to be made in a high-stress situation, and even more so if e.g. a person injury is involved.

1.3.2 System requirements

The general requirements of the decision support system are the following:

[3]One of the scenarios will (of course) be the standard one of taking no special action at all.

- All relevant information regarding the manufacturing process must be stored prior to use. This includes alternative ways of performing activities, that are not used in daily operations but can be interesting if all normal alternatives are gone.

- It must be integrated with a ERP[4]-system or the like in order to have information regarding the current state as up to date as possible, so that the user needs to input a minimum amount of data when consulting the system.

- It should be able to recognize various (unexpected) events, of which the most important are:

    - A resource becomes unavailable, either for the entire time horizon or until a certain time. This event type includes malfunctioning machines, robots or transports, operators getting ill or injured, etc. We note that in order to associate the event with appropriate special actions it is necessary to distinguish between different resource types.
    - Delayed arrivals of products or raw materials.
    - Priorities or desired completion times for products are changed.
    - New product orders are received.

  The last three items all concern the ability to change order information.

- It should be possible to define a number of possible special actions and their associated costs and effects. Examples of such actions are:

    - Calling for external service personnel that can repair errors on machines or transports faster than otherwise.
    - Hiring a replacement or additional resource. The expected delay until arrival must be specified (delays and costs of course vary with resource type).
    - Buying a (sub-) product type from an external supplier instead of producing it yourself. Delivery times etc. have to be estimated.
    - Ordering overtime. This gives in effect more working time before some of the desired completion dates.

- There should be some kind of intelligence for selecting only scenarios corresponding to special actions that are relevant for the event that occurred.

- The scheduling model used in the different scenarios should allow for a variety of details and constraints. We discuss this in more detail below.

- The total computing time should be very short, typically not more than say 10 seconds.

- The suggested scenario, the corresponding best schedule, and relevant consequences should be presented in an easily comprehensible way to the user.

[4]Enterprise Resource Planning

1.3.3 Requirements for the scheduling model

One important source of information for the scheduling problem is the so called *operation list* for a product type. It specifies the activities (called *operations*) that are to be performed when manufacturing an item of the product type, together with information on time and resource usage. In the simplest case the sequence of operations is uniquely given, and a single machine is specified for each operation together with a processing time. We will need a much more general and detailed specification than that, but it is good to have this basic case in mind when we describe a more advanced model. The minimum requirements for our scheduling system are that the following modelling features should be handled:

- The scheduling objective function should include costs for performing activities and penalty costs for late deliveries.
- Besides the processing of a product in a machine, other types of activities such as transportation of a product and setup of a machine should be possible to include in the model.
- Activities requiring more than one resource (e.g. a machine, an operator and a tool), possibly with different time usage for the resources, should be modelled adequately.
- It should be possible to treat identical resources capable of performing the same activities as a single so called *volumetric resource*. Examples of volumetric resources are parallel machines, a pool of equally qualified operators, and a set of identical tools.
- The situation with buffers having limited capacity for storing products has to be treated if necessary. The most important example for our application is machine input buffers. Most other storage places used e.g. between operations can be regarded as infinite since the operators usually can find some alternative space if necessary.
- We should be able to model in a natural way the case where we have more than one item of a product type that has to be manufactured, since we might not want to distinguish between identical items in certain situations.
- It should be possible to model alternative ways of performing activities using different sets of resources with different time usage.
- We should be able to model suitable initial conditions for the scheduling scenario such as:
    - Possibly different arrival times of products.
    - Ongoing activities.
    - Delayed availability of resources.

It is good, but not absolutely necessary, if we could model the following cases as well:

- Instead of just having alternatives for single activities, we could have product "paths" of a general fork-join type. The number of activities in different paths need not be the same.

- A resource may be tied up by a product for several successive activities, e.g. a fixture or a pallet.
- We could have activities for a product type where the order of performance is not uniquely specified.
- We could have assembly activities where two or more different product paths join into one.
- The setup time of a machine may depend on the previous activity performed. This case is called *sequence-dependent setup times.*
- We may have groups of operators with different skill levels which allow them to perform some activities but not others.

We will not include uncertainty in the scheduling model, since this is more than can be handled with a detailed model and a short execution time. However, as discussed earlier the stochastic element is not that significant with the short time horizon we have, so this simplification is well motivated.

1.4 A preliminary problem formulation

The main focus of the first stage of this project has been the modelling and solving of individual scheduling scenarios. Before we describe existing modelling techniques, scheduling theory and solution methods, we will in an informal way present one possible way of modelling and solving a scheduling scenario. Although there are many other ways of doing this for such a problem, the principle we describe is perhaps the most straight-forward one and we believe that it can be helpful to the reader to have it as a background when studying the rest of the report.

1.4.1 A discrete event model

In this model, we will explicitly follow the state of the manufacturing system in time, beginning from the starting time of the scheduling scenario. We will study the system only at certain discrete time points, namely when some kind of event has occurred. Examples of such events are:

- A product arrives at a buffer.
- A resource becomes available.
- An activity is finished.

If we define such events properly, we can restrict all decisions regarding "what to do next and when to start doing it" to these time points plus of course the starting time point of the scenario. This removes the continuous element of assigning start times to activities, and we will still be able to find an optimal schedule[5]. This can be motivated by saying that it makes no sense to postpone the starting of an activity unless we are waiting for another event to occur. We now temporarily define the word *action* to include information on

[5]It is possible to define strange objective functions so that this does not hold, but our cost-based function is well behaved in this manner.

- An activity that is to be started.
- The assignment of resources to this activity (if there are alternatives).
- The starting time (which is the time of an event or the starting time of the entire scenario).

The scheduling process is now to select a sequence of actions in ascending order of the starting time until we have reached a desired final state, without violating any constraints of system. If our system is specified properly, we can make an important observation: Given the initial conditions of the scheduling scenario, a time point and an allowed sequence of actions up to this point, the state of the system at that time point is uniquely defined. Such a system belongs to the class of *discrete event dynamic systems.*

1.4.2 Exploring the decision tree

But how do we select an action sequence? There are a large number of possible sequences and we want to find the best, or at least one that is good enough for our purposes. One way of implicitly organizing all possible sequences is in a *decision tree.* A node in the tree corresponds to a reachable state of the system, and the node has one branch leading to a child node for each action that is possible to take next. A node without any child nodes (a so called *leaf*) corresponds to a final state where all activities have been completed or to a non-final state from which it is impossible to make any more actions. The root node of the tree is the initial state.

Each path from the root node to a leaf node corresponding to a final state is thus associated to a unique action sequence for which we can calculate the cost. Recall that we defined the cost of the schedule to be the sum of costs for performing activities and costs for late deliveries. The scheduling problem is now to find such a path that has as low cost as possible. But one problem is that we do not have this tree explicitly given in a scheduling scenario, instead we have to build it gradually starting from the root. It is not difficult to realize that the size of the decision tree becomes huge as the action sequence required to reach a final state gets longer. This means that for larger problems, we cannot hope to explore the entire tree, instead we will have to focus on some interesting parts and settle for the best path sequence we can find there. An important task is thus to make this search procedure as fast and intelligent as possible.

Chapter 2

Graphical modelling techniques

Before we discuss solution methods for scheduling problems, we will take a look at some graphical tools that can be used for modelling manufacturing systems. When designing a system like ours that will be used by non-experts, the advantage of using a model which can be viewed graphically is obvious. If the graphical representation follows a wide-spread standard, it is of course even better. Therefore, an important part of the first phase of this project has been to evaluate some established graphical models with respect to our application. A reader only interested in parts directly connected to scheduling may settle with reading Sections 2.1 and 2.4 only.

2.1 Petri nets

Petri nets is a graphical and mathematical tool for the modelling of discrete event systems, based on the early ideas of Petri (1962). The research area of Petri nets is very large and we will only give a short introduction to the subject here, to serve as a background when we discuss the application of Petri nets to scheduling problems in subsequent chapters. We refer to the literature for further reading on general Petri net theory. A classic book on the subject is Peterson (1981), and a likewise well-cited reference is a paper by Murata (1989). One of the major applications of Petri nets is the description of manufacturing systems. The book by DiCesare et al. (1993) focuses on this.

2.1.1 A simplified Petri net

A Petri net can be viewed as a directed graph with two classes of nodes, denoted *places* and *transitions*. There are two kinds of directed arcs in the graph. An arc of the first kind goes from a place to a transition and is called an *input arc* of the transition. The place is referred to as an *input place* of the transition. The second kind of arc goes from a transition to a place and it is referred to as an *output arc* of the transition. The place is accordingly called an *output place* of the transition. Since no arc connects a place to a place or a transition to a transition, the graph is *bipartite*. Graphically, places are drawn as circles, and transitions are drawn as bars or boxes (we will use the latter). An example of a simple Petri net is shown in Figure 2.1.

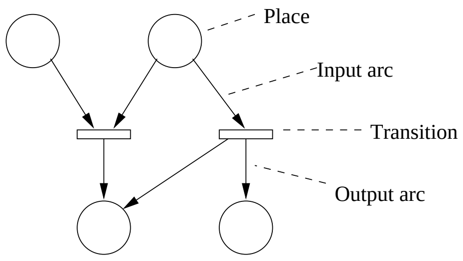

Figure 2.1: An example of a Petri net.

In the Petri net, places represent *states* and transitions represent *events* (or *actions*). The state of the entire system is determined by a *marking* of the net, which is a distribution of so called *tokens* in the places. Tokens are graphically drawn as dots inside place circles. In the simplest case, there can be at most one token in each place in the net. A place with a token in it is called *marked*. A place represents a boolean property of the system (e.g. the idleness of a machine) which is true if the place is marked and false otherwise.

The state of the system (i.e. the marking of the Petri net) can change by the so called *firing* of a transition. A transition is said to be *enabled* (or *ready to fire*) if and only if all input places of the transition are marked and all output places are unmarked. When the transition is fired, the new marking of the Petri net is obtained by first removing all tokens in the input places and then adding a token in each output place. This is illustrated in Figure 2.2. The evolution of the marking in the Petri net through the firing of transitions is sometimes called the *token game*.

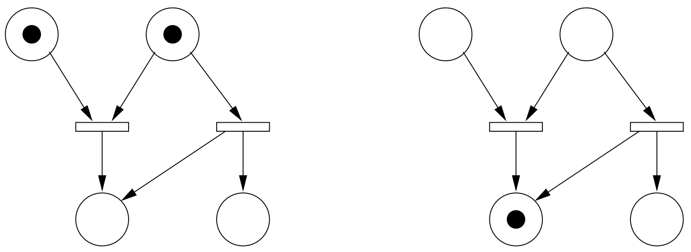

Figure 2.2: Left: A marking of the Petri net of Figure 2.1. Right: The new marking obtained when the leftmost transition is fired.

The marking that describes the initial state of the system is referred to as the *initial marking*. The structure of the Petri net and the initial marking govern the states (markings) that are reachable by firing a sequence of transitions. These states and the ways of reaching them are often analyzed in terms of the so called *reachability graph* (not to be confused with the Petri net itself). In this graph, there is a node for each marking that can be reached from the initial marking (including the initial marking itself). There is a directed arc from a node A to a node B if and only if the marking corresponding to B can be reached from the one corresponding to A by the firing of an enabled transition. In general, there can be many paths in the graph between two nodes. As we will see later, solving a scheduling problem modelled with a Petri net is equivalent to finding an optimal path between the node corresponding to the initial marking and a node corresponding to a desired goal marking. One never works explicitly with the reachability graph however, instead one explores only parts of it implicitly. The reasons for this are that the reachability graph is not known before a solution process starts, and that it in most cases is way too large to store in computer memory.

2.1.2 The standard Petri net

The simple type of Petri nets described in the previous section can be extended to a more general version (which we will refer to as a *standard* Petri net), where we allow any number of tokens in a place and multiple arcs between a place and a transition or vice versa. The latter extension means that we have a *multi-graph*[1]. We now say that a transition is enabled if and only if for each input place of the transition there is at least as many tokens as there are input arcs between the place and the transition. Note that we now do not have any conditions on the output places. When the transition is fired, from each input place of the transition a number of tokens equal to the number of input arcs between the place and the transition are removed, and to each output place a number of tokens equal to the number of output arcs between the place and the transition are added. We illustrate this in Figure 2.3.

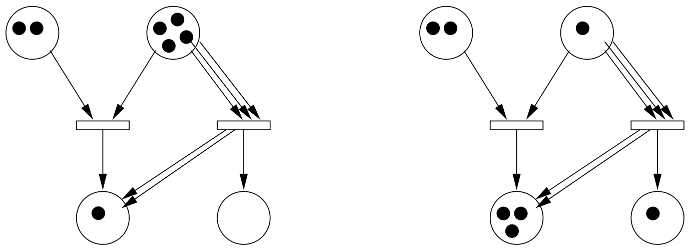

Figure 2.3: Left: A marking of a Petri net of the more general type. Right: The new marking obtained when the rightmost transition is fired.

2.1.3 An algebraic description of the system

If a place is both an input place and an output place of the same transition, we say that we have a *self-loop*. The structure and dynamics of a Petri net without self-loops can be compactly described in matrix form. In the *input matrix* I rows correspond to places and columns to transitions. The entry on row i and column j in I is equal to the number of input arcs going from place number i to transition number j. We analogously define the *output matrix* O. The *incidence matrix* C is defined by $C = O - I$.

A marking can be described by a column vector M where element i is equal to the number of tokens in place number i. Let E_j be the *characteristic vector* of transition

[1]We note that it is also common to use an equivalent model based on an ordinary graph but with *weights* on the arcs, with the weight on an arc equal to the number of parallel arcs in our model.

number j, i.e. element j of E_j is 1 and all other are 0. We can now describe the transition enabling rule: transition number j is enabled if and only if

$$M \geq IE_j. \tag{2.1}$$

Assume now that we have two markings represented by the vectors M_1 and M_2. It can be shown that M_2 can be obtained by firing transition j at M_1 if and only if the following holds:

$$M_2 = M_1 + CE_j \geq 0. \tag{2.2}$$

Assume now that marking M_k is reachable by firing a sequence of transitions starting from marking M_0. We introduce the *firing count vector* N in which element j equals the number of times transition number j was fired in the sequence. The following equation, often referred to as the *Petri net state equation*, now holds:

$$M_k = M_0 + CN \geq 0, \; N \geq 0. \tag{2.3}$$

Unfortunately, the relation between this equation and a firing sequence does not go both ways, as it did for the firing of one transition. Given two markings M_0 and M_k we can find a vector N which solves (2.3), where there is no possible firing sequence with firing count given by N so that M_k is reached from M_0. Such an N is called a *spurious solution* of the Petri net state equation. The existence of spurious solutions makes this algebraic approach less powerful than it otherwise would be when analyzing properties of Petri nets.

2.1.4 Extensions of standard Petri nets

A natural extension of the standard Petri net we have described above is to include time in the model. There are two widely accepted approaches for doing this, *place-timed* Petri nets and *transition-timed* Petri nets, but other variants exist as well. In these models the token distribution alone is not sufficient to fully describe the system, and one has to take into account the current time of the system when analyzing e.g. which transitions are enabled.

In a transition-timed Petri net, tokens are removed from the input places when a transition is fired, but new tokens are not introduced into the output places until a certain time has passed in the system. We thus associate time delays to transitions. In a place-timed Petri net, the firing of transitions are immediate, but a token introduced in an output place must wait for a specified time before it can be used as input for a transition. In this case time delays is associated to places, and the places with non-zero time delay are often referred to as *timed places.* Usually the net is constructed in such way that there can be at most one token in a timed place. It has been shown that place-timed and transition-timed Petri nets are equivalent to each other with respect to modelling power.

Another extension is to let individual tokens in a place be distinguishable by giving them a "color". The input and output arcs of a transition are labelled with formal expressions, constraining the color combinations of tokens in input places that are allowed when firing the transition, and determining the color of the new tokens introduced in the output places. These kind of nets are called *Colored Petri nets.* Nowadays, Colored Petri nets have been generalized to allow arbitrary complex data types for tokens and to let expressions for arcs and transitions be constructed with a special programming language. Colored Petri nets belong to the class of *High-Level Petri nets.* An introduction to the subject can be found in e.g. Jensen (1997).

2.1.5 Advantages and disadvantages of Petri nets

The advantage of Petri nets over many other graphical modelling tools is that it has a mathematical formalism that makes the dynamic behaviour of the underlying system well-defined, and amenable to theoretical analysis using results from e.g. linear algebra and graph theory. If an intuitive and easy-to-read graphical model is the main concern rather than an exact description of the underlying system, Petri nets have some drawbacks however. First of all, even if places and transitions are labelled with names of the corresponding states and events, a reader must learn Petri net basics before he can understand the model. Secondly, Petri nets tend to become graphically very muddled for all but small examples. There have been some suggestions for a hierarchical composition of Petri nets in order to partially overcome this drawback, but such techniques are mainly used in the construction phase of the modelling.

2.2 IDEF0

IDEF0[2] is a tool for modelling a variety of automated and non-automated systems, and can be used for the specification and implementation of a new system and for analysis of an existing system. IDEF0 was developed in the 1970s within the U.S. Air Force Program for Integrated Computer Aided Manufacturing (ICAM), and is now adopted as a Federal Information Processing Standard (FIPS) by the National Institute of Standards and Technology. We will here only briefly describe some IDEF0 basics, for more details we refer to Nat (1993).

2.2.1 Graphic syntax

Whereas Petri nets are mainly status oriented, IDEF0 is a more function oriented graphical modelling technique. An IDEF0 model is composed of a hierarchical series of *diagrams*, which can be of three types: *graphic*, *text* and *glossary*. The graphic diagrams define system functions and functional relationships, and the text and glossary diagrams provide additional information. There are two main graphic components in the IDEF0 syntax: *boxes* and *arrows*.

Boxes represent *functions*, which are defined as activities, processes or transformations. The box has a number, uniquely identifying it within the diagram. Arrows represent data or objects related to functions. Arrows start and end either at the side of a box or at the boundary of the diagram. An arrow which has one of its ends at diagram boundary is called a *boundary arrow*, whereas an arrow going between two boxes is called an *internal arrow*. Arrows may also fork and join. The role of an arrow with respect to a function is determined by which side of the box it is connected to, and by its direction. The possible connections of arrows and boxes are shown in Figure 2.4.

Arrows entering the left side of the box are *inputs* to the function, which can be transformed or consumed by the function to produce *outputs*, which are arrows leaving the box at its right side. Arrows entering the top side of the box are *controls*, which specify the conditions required for the function to produce correct outputs. Arrows attaching to the bottom side of the box are *mechanisms*. Those pointing upwards identify the means that support the execution of the function. A mechanism arrow pointing downwards is called a *call arrow*, and it can be used for sharing detail

[2]Integration DEFinition language 0.

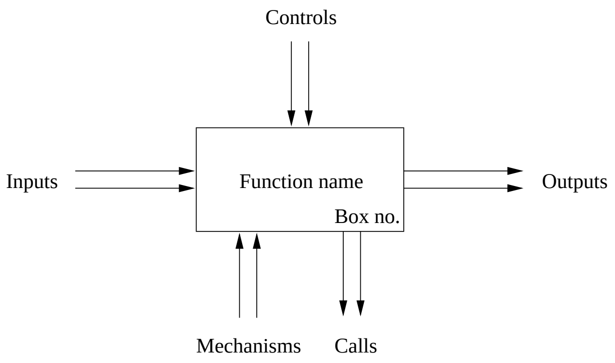

Figure 2.4: An IDEF0 box.

between different models or between portions of the same model. Note that all these definitions are with respect to a certain function box. An arrow can have different roles at different boxes, e.g it can be the output of one function box and the control of another.

In our case, a function box on a detailed diagram level could represent a manufacturing activity, e.g. the processing of a product in a machine. The product would be both input and output, and if we have an assembly activity there could be additional inputs for subproducts or raw materials. The machine and any other resource needed for the activity would be mechanisms. Controls could represent the constraints concerning the activity.

2.2.2 Hierarchical decomposition

A box on a diagram may be decomposed into a *child diagram*, which typically contains several boxes representing sub-functions of the decomposed function. The box is then referred to as a *parent box*, and the corresponding diagram is called a *parent diagram.* A parent box has a label below it stating the name of the child diagram. The arrows connecting to the parent box are the boundary arrows of the child diagram. An example of decomposition is shown in Figure 2.5.

In this way, diagrams are arranged in a tree structure. All IDEF0 models are required to have a so called *context diagram* named A-0 at the top level (the root of the tree), containing a single function box. Context diagrams relate the model to its environment. If desired, one can also have a tree of context diagrams, where A-0 is one of the leafs (and the root for the actual model diagram tree).

2.2.3 Advantages and disadvantages of IDEF0

As we have seen, the basics of an IDEF0 model are defined by only a handful of components and rules[3] and are thus quite easy to learn. Also, the IDEF0 diagrams are very intuitive, even for a person who are not familiar with the model. The hierarchical decomposition makes models well-structured and provides a good overview for a reader.

[3]We note however that there are more components, rules and recommendations in the standard, we have now only presented the most essential parts here.

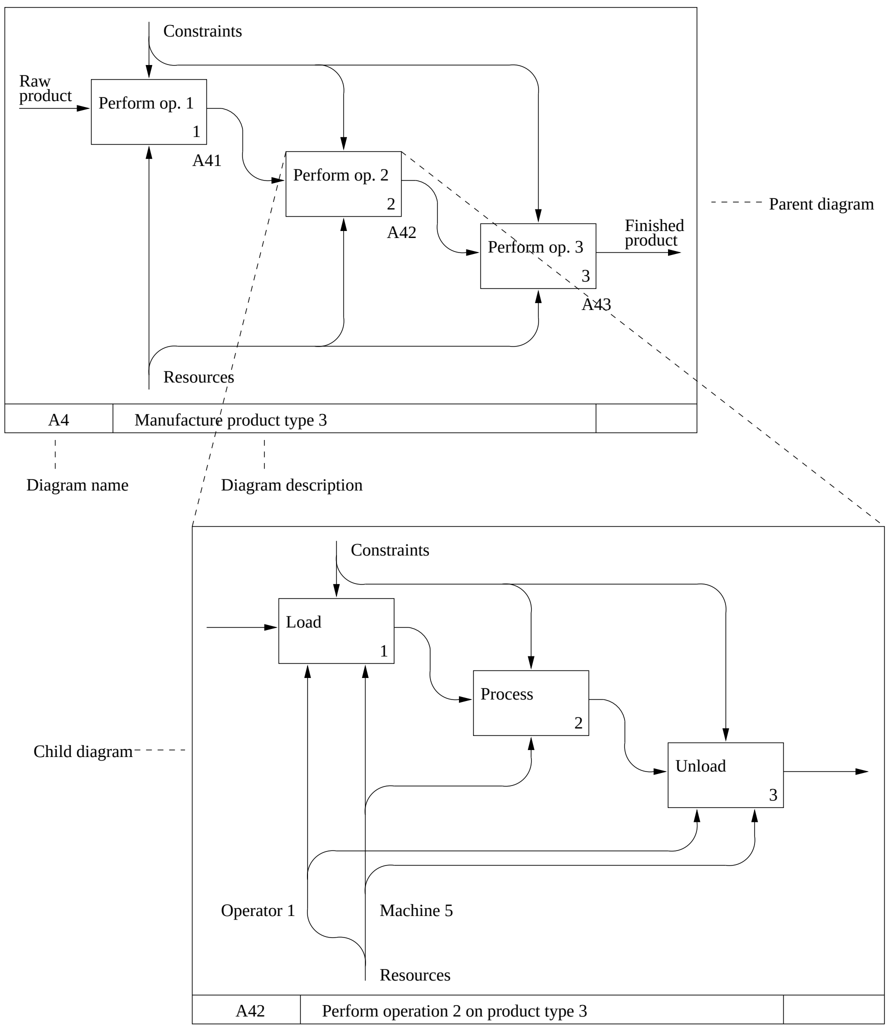

Figure 2.5: The decomposition of a parent box into a child diagram.

From our point of view, a drawback is that objects and object states are not well-defined in the model, nor is the dynamic behaviour of the system. This means that we would have to impose additional rules and interpretations on the IDEF0 model in order to be able to base a scheduling algorithm on it, something one of course wants to avoid if possible. Furthermore, we see no obvious way of doing such an extension. We also note that there are some IDEF0 rules that would be difficult to follow in a general scheduling model, one is e.g. that a graphic diagram should have no less than three and no more than six function boxes (the only exception being the A-0 context diagram).

2.3 IDEF3

The IDEF3 Process Description Capture Method is another graphical technique in the IDEF-family that can be used for describing manufacturing systems. We will briefly present some IDEF3 basics here, for more detailed information we refer to Mayer et al. (1995). IDEF3 is mainly intended for facilitating the description of an existing system. Mayer et al. (1995) makes a distinction between the words *description* and *model*, saying that "descriptions record knowledge that originates in or is based on observations or experience" and that "a model constitutes an idealized system of objects, properties, and relations that is designed to imitate, in certain relevant aspects, the character of a given real-world system". IDEF3 allows for incomplete descriptions of a system, and alternative descriptions of the same process in the system. We are however interested in using IDEF3 components for making a well-defined model.

2.3.1 Basic graphic components

We will start with defining some important concepts in IDEF3. A *process* is an ordered sequence of events. A *scenario* describes the setting within which one or more processes occur. IDEF3 has both a *process-centered* view and an *object-centered* view. In the process-centered view, one organizes process knowledge with a focus on processes and their temporal, causal and logical relations within a scenario. In the object-centered view, one organizes process knowledge with its focus on objects and their state change behaviour in a single scenario or across multiple scenarios. Graphically, these views are presented in *process schematics* and *object schematics.* An object schematic that is concerned with a single scenario is called a *transition schematic.*

Process schematics

In the process schematics, the *unit of behaviour (UOB)* box is the most important component. It represents an event, a decision, an action or an entire process. UOB boxes are connected with *links* shown as arrows. The links define precedence relations or other constraints. Links can fork and join at so called *junctions* (smaller boxes). Just like IDEF0, IDEF3 allows for an hierarchical description. A UOB can be *decomposed* into a more detailed description. This is called a *decomposition schematic.* Figure 2.6 illustrates all these concepts.

An *instance* of a UOB is an occurrence of the process it represents. An *activation* of a schematic is a collection of instances of some or all of the UOBs whose temporal and logical properties satisfies the conditions specified in the schematic. A simple precedence link, as the ones shown in Figure 2.6, does not force instances of both UOBs it connects to occur in an activation. There are a number of *constrained precedence links* in the IDEF3 semantics which constrain UOB instances in an activation in various ways, e.g. saying that if an instance of UOB1 occurs in an activation, then so must an instance of UOB2. Junctions also control the activation logic. There are five different junction types in process schematics: *AND*, *Synchronous AND*, *OR*, *Synchronous OR*, and *Exclusive OR (XOR).* In Figure 2.6, we saw examples of XOR-junctions. We will not go into details on activation rules here, see Mayer et al. (1995) for this.

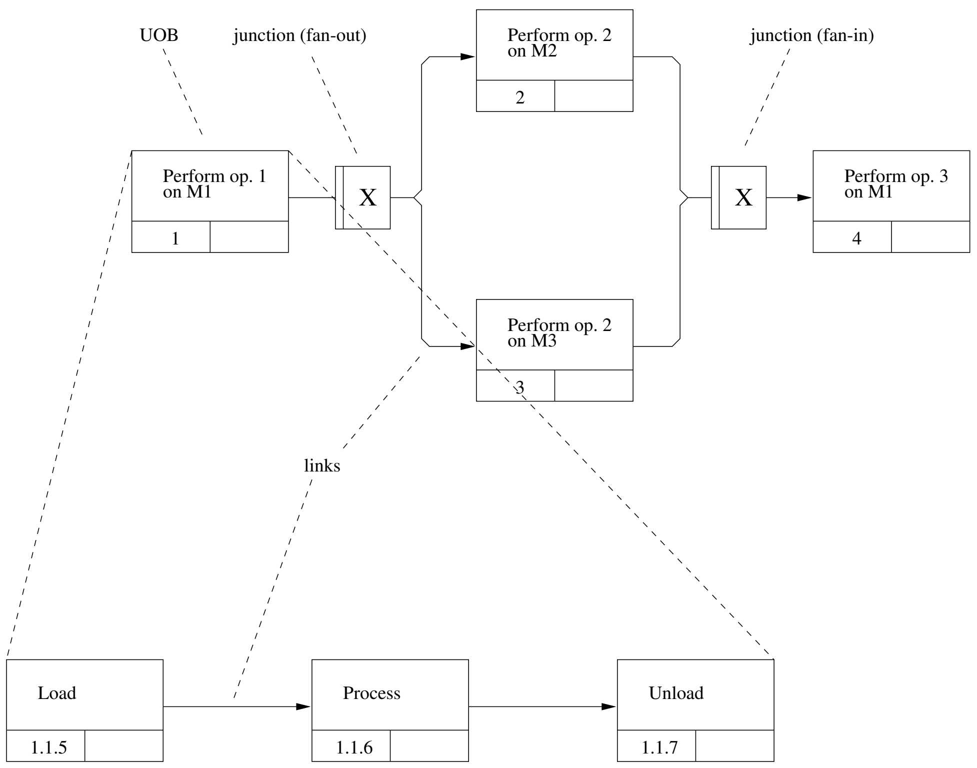

Figure 2.6: An IDEF3 process schematic with decomposition.

Object schematics

We will here only discuss the basic components used in the simplest form of object schematics: the transition schematic. In a transition schematic, a certain kind of object being in a certain state is represented by a circle with a label specifying both the kind and the state. *Transition links* connect such *object state symbols. Junctions* (small circles) can be used when more than two object state symbols should be connected. A junction's logical meaning can be either AND, OR or XOR. A simple example is shown in Figure 2.7.

Referents

A transition schematic does not give much information in itself regarding the function of a system. By attaching a *referent* to a transition link or an object state symbol, the transition schematic can be related to a UOB, to a scenario or to another transition schematic. Referents, which are represented as boxes, can be either of type *call-and-continue* or *call-and-wait.* The type affects the timing of transitions. In Figure 2.8, it is shown how the transition schematic example is connected to a UOB with a call-and-wait referent. In this example, an instance of the UOB must terminate before the object state transition can occur. Referents can also be used in process schematics, e.g. for connecting to a transition schematic.

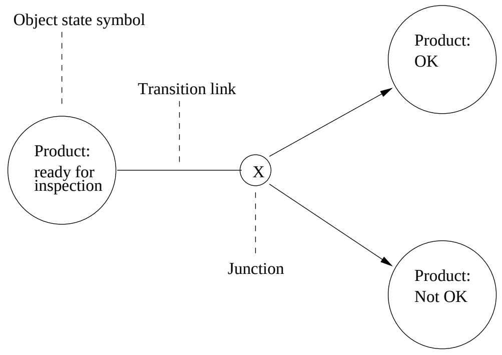

Figure 2.7: An IDEF3 transition schematic.

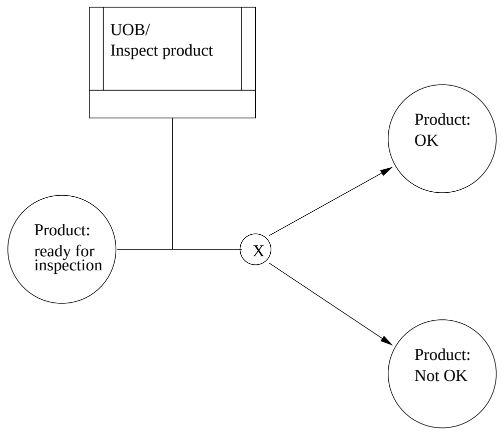

Figure 2.8: The example transition schematic connected to a UOB.

2.3.2 Other IDEF3 components

IDEF3 includes many more components than we have presented here. For example, there are object schematics called *first-order schematics* and *second-order schematics*. They can be used for describing properties of objects and relations between objects which are not connected to state transitions. There are also so called *elaborations* that can be attached to schematic elements, having for example an additional textual description. If desired, one can use a special well-defined *elaboration language*. It is similar to a *logic language*, a subject we briefly touch in Section 4.2.

2.3.3 Advantages and disadvantages of IDEF3

IDEF3 is not as intuitive and easy to learn as IDEF0, but it offers a much richer syntax. The range of systems and situations that can be described is much larger than for IDEF0 or Petri nets. Unlike IDEF0, the dynamic aspects of a system can be modelled to some extent, but not as precise as in Petri nets. Although IDEF3 must be regarded as a very powerful tool in general, we believe that it is not particularly well-suited as the basis for a scheduling algorithm.

2.4 Project planning techniques

In the area of project planning, graphical models have been used to describe and solve simple scheduling problems for almost forty years. We will briefly review two similar graphical models here, the *potential task graph* and the *PERT*[4] *graph*. We do this not because they are likely to be extendible to our problem, but to provide the reader with some background on the connection between project and production scheduling problems. More advanced project scheduling problems are often similar to problems encountered in manufacturing. In fact, the potential task graph model can be modified and extended to the so called *disjunctive graph*, which is a commonly used model in production scheduling. We will discuss the latter in Section 4.1.3.

We will in the following presentation follow the book by Gondran and Minoux (1984). The basic problem which the potential task graph and the PERT graph are designed to model is the *central scheduling problem*. In this we have N *tasks* (activities), where task i has a duration d_i. The goal of the problem is to assign start times to the tasks in order to minimize the maximum completion time of any task, while obeying a set of so called *precedence constraints*. A precedence constraint between tasks i and j means that task j are not allowed to start before task i has been finished.

2.4.1 Potential task graphs

In the potential task graph, the tasks are represented by nodes. There is a directed arc between nodes i and j if there is a precedence constraint between tasks i and j (assuming the same numbering on nodes and tasks). This arc has a length d_i. We also add two fictitious tasks of duration 0: an initial task which has to precede all others and a final task which all other tasks precede. We will illustrate this with an example from Gondran and Minoux (1984), which is specified in Table 2.1. The potential task graph for the example is shown in Figure 2.9. Note that a potential task graph is always acyclic.

Task	Name	Duration	Previous tasks
1	A	7	-
2	B	3	A
3	C	1	B
4	D	8	A
5	E	2	D, C
6	F	1	D, C
7	G	1	D, C
8	H	3	F
9	I	2	H
10	J	1	E, G, I

Table 2.1: A simple central scheduling problem.

It can be shown that the minimum duration of the project is equal to the length of a longest path in the graph. Such a path is called *critical*. Given this minimal project duration, it is easy to calculate the earliest and latest starting times for the

[4]Project Evaluation and Review Technique.

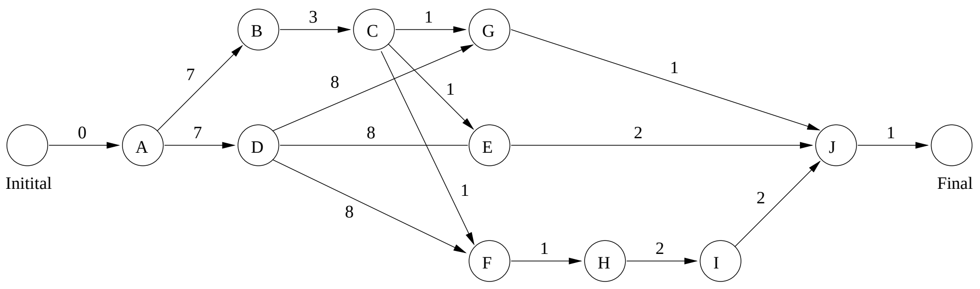

Figure 2.9: A potential task graph.

tasks. The difference between the latest and the earliest starting times are called the *slack* of the task. The tasks with zero slack are called *critical tasks*, and they always correspond to nodes lying on a critical path. If a critical task is delayed, so will the entire project be. In the example above, there is one critical path: A-D-F-H-I-J.

The potential task graph can be extended to model some other constraints by adding arcs with certain lengths. For example, if task i cannot be started before a time t_i, we can add an arc from the initial node to node i having length t_i.

2.4.2 PERT graphs

In a PERT model, the project is divided into a number of *stages*. A stage is defined by a set of tasks already carried out and it is represented in the graph as a node. We have an initial stage where no tasks have been performed, a final stage where all tasks are done, and a number of intermediate stages. A task i is represented by a directed arc in the graph, going between two nodes (stages) and having a length d_i. If there is a precedence constraint between tasks i and j, the end node of i should be the start node of j. There can be more than one arc between two nodes, so the graph is in fact a multigraph. The graph will be acyclic. We show the PERT graph of the example from 2.1 in Figure 2.10.

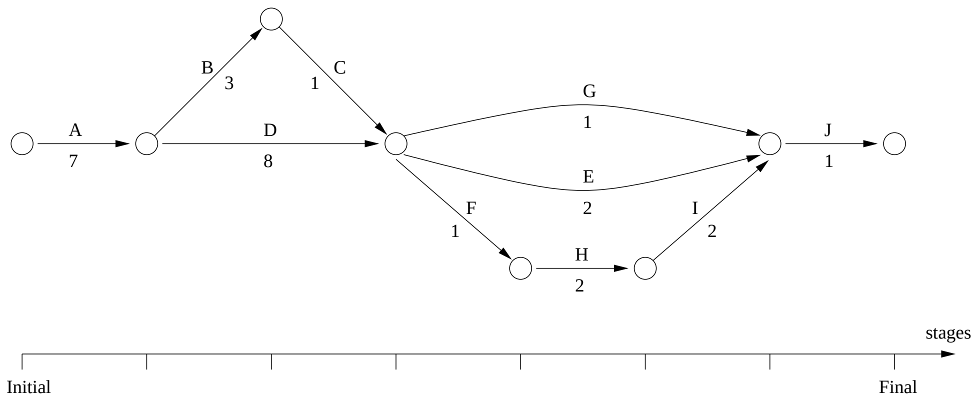

Figure 2.10: A PERT graph.

As can be seen, it is not as straightforward to build a PERT model from a given central scheduling problem as it is for the potential task graph. There are however other problem formulations for which the PERT model is natural. As for the potential task graph, it is easy to find the minimum project duration and the critical tasks in the PERT graph. The PERT graph is however more difficult to extend with additional constraints.

Chapter 3

Optimization and Scheduling theory

In this chapter we will present some terminology and concepts of scheduling theory. We will also present some well-known scheduling problems that are of interest in manufacturing. As mentioned in Section 1.1, scheduling is a broad area with many applications. Some of the more basic problems can arise not only in manufacturing but also in e.g. scheduling of computer processes and project planning. As we add more features specific to the application areas, the similarities between the resulting scheduling problems decrease. Therefore, examples and methods from other application areas will not be that interesting for us and we will only mention a few examples in this report.

3.1 Optimization problems

We will in this report use some standard terminology for optimization problems. For readers unfamiliar with this, we present some definitions here. An *optimization problem* is the task of minimizing or maximizing an *objective function* under a set of *constraints*. In the simplest case, the objective function is constructed from a set of variables, called *decision variables*, and the constraints are equations formed by these variables. There are also many cases where the objective function and the constraints cannot be explicitly stated using mathematical formulae. We will for simplicity use the former case as example when explaining some of the following definitions, but they apply to the general case as well.

The problems of interest here are called *combinatorial optimization problems*, which in the simplest case means that some or all of the decision variables take only discrete values. Usually, an optimization problem of a particular type is generally stated, with a set of parameters controlling the number of variables and constraints and the exact definition of the objective function and the constraint equations. For a given assignment of values to these parameters we say that we have an *instance* of the problem. Note that it is very common to only use the word problem when it is clear from context that one refers to a problem instance.

For a given optimization problem instance, an assignment of values to the variables such that all constraint equations hold is called a *feasible solution* (or just *solution*). An assignment for which one or more constraints are violated is called an *infeasible* *solution.* The value of the objective function for a given variable assignment is called the *objective value.* A solution for which it is not possible to find another solution with better objective value is called *optimal.* Note thus that the word solution for an optimization problem (instance) does not imply "best" with respect to the objective function, it is only connected to the constraints.

A *solution method* (or *algorithm*) for an optimization problem is a well-defined procedure for finding a feasible solution (as good as possible) to the problem. If the method guarantees to find an optimal solution for any problem instance given enough computing time and memory, we say that the method is *exact*, otherwise we call it *approximate* or *heuristic.*

3.2 Scheduling glossary

Here we present a list of terms (in alphabetic order) frequently used in the scheduling literature. Throughout the report, we will use some of these terms without further explanation, so the reader is urged to return to this section if necessary. In the descriptions below, we will use italics for terms that have their own explanation elsewhere in the list.

Activity: A general term for the tasks that are to be performed by a manufacturing system. Frequently, the activities are grouped into *jobs*, in which case the individual activities of the job often are referred to as *operations.* Activities require the availability of *resources* in order to be performed. The term *task* is sometimes used instead of activity.

Allowance: The difference between the *due date* and the *release date* of a *product* (*job*), i.e. the maximum desired *flow time* for the product.

Alternative paths: See *routing flexibility.*

Arrival time: See *release date.*

Assembly: The process of creating one product out of two or more sub-products.

Batch: The term for when we have more than one item of a *product* type that should be manufactured. See also *lot size.*

Buffer: A *resource* that serves as a storage facility for one or more *products.*

Capacity: A measure of how much a *resource* can be utilized[1]. A common example of specifying capacity is the number of items available of a certain resource over time.

Completion time: The time when a *product* (*job*) is finished. The term can also be used for individual *activities* (*operations*).

Deadline: See explanation at *due date.*

Deadlock: The situation when two or more *products* are in a cyclic wait state due to limited or no *buffer* space. If no product can continue its processing we talk of a global deadlock.

[1]We note that the word *capacity* is sometimes used with a slightly different meaning. Our definition corresponds to what is then called *maximum capacity.*

Disjunctive constraint: A constraint between a pair of *activities* (*operations*) saying that they cannot overlap in time, usually because the *resource capacity* is not large enough for both activities at the same time.

Due date: The time when a *product* (*job*) is estimated/desired to be finished. The due date is a "soft" constraint that can be violated in a schedule, but at a penalty cost. In addition we can have a "hard" constraint that the product must be finished before a certain time. This time is called the *deadline* and is of course larger than or equal to the due date.

Earliness: This non-negative quantity is equal to the difference between the *due date* and the *completion time* of a *product* (*job*) if the product is finished before the due date, and 0 otherwise.

Flow time: The difference between the *completion time* and the *release date* of a *product* (*job*), i.e. the amount of time the product spends in the system. Optimization objectives based on flow times are used when it is desirable to minimize in-process inventory.

Job: Corresponds to the manufacturing of a certain *product.* The *activities* needed to finish the product are called *operations.* Usually, the operations within a job cannot overlap in time.

Just-In-Time (JIT): An optimization objective where it is desired that *products* are neither finished too late or too early relative to the *due date.*

Lateness: The difference between the *completion time* and the *due date* of a *product* (*job*). Note that this quantity is negative if the job is completed before the due date.

Lot size: The number of items that are to be produced of a certain *product* type (in the same time period). See also *batch.*

Machine: The most common type of *resource.*

Makespan: The maximum *completion time* of any *activity* considered in the scheduling problem. If activities are grouped into *jobs*, the makespan is the last completion time of any job (*product*). Perhaps the most common optimization objective used, at least in the academic society. Also called *schedule length.*

Operation: An *activity* of a job. Usually an operation refers to the processing of a (not yet finished) *product* in a machine, but sometimes other activities such as transports, setup of machines etc. are considered to be individual operations as well.

Operator: A human *resource* that operates other resources such as machines and transport devices.

Precedence constraint: A constraint between a pair of *activities* A and B stating that B cannot be started until A is finished. This is typically used to specify the order of *operations* within a *job.*

Preemption: The processing of an *activity* (*operation*) is interrupted and resumed at a later time. In most situations this is not allowed, in which case one refers to the manufacturing process as non-preemptive.

Processing time: The time it takes to perform an *activity* (*operation*). In most cases it is a fixed time independent of how and when the activity is performed, but it can also depend on the resources used (see *routing flexibility*) or other factors.

Product: The physical entity that is manufactured. Note that we define the product with respect only to our scheduling problem. In a larger setting, it might be just a sub-product.

Ready time: See *release date.*

Release date: The time when a *product* (*job*) gets available for manufacturing. Other names for this are *arrival time* or *ready time.* In some cases one talks of release dates for individual *activities* as well.

Resource: In the simplest case (called a unary resource), it is a single *machine*, *operator*, robot, tool, *buffer* etc. needed for performing an *activity* (*operation*). A volumetric resource can perform more than one activity at a time (i.e. it has a *capacity* greater than one). Examples of the latter are parallel machines, a pool of equally qualified operators, a buffer with room for more than one product etc. Other types of resources can also be specified, but these are the two most common.

Routing flexibility: In the simplest case, this denotes the situation where there can be alternative ways of performing *activities* (*operations*), using different *resources* and possibly resulting in different *processing times.* In a more general setting, routing flexibility means that there can be alternatives for entire sets of activities (operations), with possibly different number of activities in different sets. We also refer to this term as *alternative paths.*

Schedule length: See *makespan.*

Slack: The difference between the remaining time to the *due date* and the remaining *processing time* of a *product* (*job*) at a certain time. A negative slack means that the product will inevitably be finished later than its due date.

Start time: The time when an *activity* (*job*) is started. One can also talk about start time for an entire *job*, meaning the start time of its first *operation.*

Tardiness: This non-negative quantity is equal to the difference between the *completion time* and the *due date* of a *product* (*job*) if the product is finished after the due date, and 0 otherwise. Optimization objectives based on tardiness are used in situations where it is desirable to avoid late deliveries.

Task: See *activity.*

3.3 Scheduling theory

In the first chapter, we gave an introduction to the area of scheduling, and we have in the previous section defined a number of scheduling terms. We will now go deeper into some parts of scheduling theory that we will need for the following parts of this report. We will also give several references for further reading.

3.3.1 Classification of scheduling problems

We saw one way of classifying scheduling problems in Section 1.1. Here we will present some other ways of dividing problems into different types.

Classes based on the degree of certainty in the problem

The first distinction we will make is between *deterministic* and *stochastic* scheduling problems. In the former class, all parameters in the problem are known without uncertainty, whereas this is not the case in the latter.

Another classification is *static* vs. *dynamic* scheduling problems. In static scheduling problems, all jobs (or more generally, activities) that are to be completed within the time horizon of the scheduling problem are known. In dynamic scheduling problems, new jobs (activities) can be added during the scheduling time period. Note that a dynamic problem is by nature stochastic, but that static does not imply deterministic. In the dynamic case, the schedule cannot be completely finished before execution, instead it has to be constructed in stages as time passes. Therefore, it is more difficult to talk about optimality for the problem. For most real-world dynamic problems, different scheduling strategies have to be evaluated by simulations. As was discussed in the introduction, we will in this report mainly concentrate on static deterministic problems.

Complexity classes

One can also classify a scheduling problem (or any optimization problem for that matter) according to how difficult it is to solve it optimally. For some problems, so called *polynomial* solution algorithms exist. This means that the computing time the algorithm needs for any problem instance is bounded by a polynomial function in the *size* (which is measured as the length of the input data string given to the algorithm) of the instance. If the order of the polynomial is low (preferably linear), the solution algorithm is usually fast also for large problems.

Other problems can be proved to belong to the class of so called *NP-hard* problems, for which it is believed that polynomial algorithms cannot be found. Instead, the solution time for an exact algorithm will typically grow exponentially with the problem instance size. For our kind of problems, this effect is related to what is sometimes called the *combinatorial explosion.* In practice, larger instances of an NP-hard problem are not possible to solve optimally, instead one has to settle for an approximate solution. There are also so called *open* problems for which no polynomial algorithm has been found but where NP-hardness remains to be proved.

This research area is often referred to as *complexity theory*, and a good general book on the subject is the one by Garey and Johnson (1979). For deterministic scheduling problems, the book by Brucker (1998) has a very extensive list of polynomially solvable problems and NP-hard problems. However, we need not concern us much about these issues, since almost all scheduling problems of practical interest are NP-hard. This is particularly true for the applications of interest in this project. Also, the size of real-world problem instances is usually so large that they cannot be solved optimally.

More on deterministic scheduling problems

In the academic society, the by far most studied class of problems are (static) deterministic scheduling problems. Some recent books having an extensive treatment of the subject are Pinedo (1995), Błażewicz et al. (1996) and Brucker (1998). Some of the newest contributions to deterministic scheduling theory are discussed in Lee et al. (1997). In all of the above mentioned books, a well established formal notation for further classifying deterministic scheduling problems is used. We will however not use it in this report, since our scheduling problem falls outside this classification scheme.

Even if this is the simplest class of scheduling problems, all but the more basic problems are extremely difficult to analyze and solve. We will later in this chapter give several examples of commonly studied deterministic scheduling problems, and in the next chapter we will review different types of solution methods.

More on stochastic scheduling problems

Stochastic scheduling problems belongs to the research field of *optimization under uncertainty*, or *stochastic programming* as it is often called. In many cases, one tries to optimize the expected value of the objective function. A general treatment of the subject can be found in e.g. the book by Birge and Louveaux (1997). Apart from a few simple cases, scheduling problems with uncertainties in e.g. processing times are generally much more difficult than their deterministic counterparts. The range of problems that allow any deeper theoretical analyses to be made are even narrower than in the deterministic case. The book by Pinedo (1995) has several chapters on stochastic scheduling. A collection of articles on both deterministic and stochastic scheduling can be found in Chrétienne et al. (1995). We particularly mention the article by Weiss (1995a), which is a tutorial on stochastic scheduling.

3.3.2 Types of schedules

With a *schedule* we mean a feasible solution to the scheduling problem. We will now discuss some classes of schedules. If there are decisions left to make in the scheduling process, we speak of a *partial* schedule. If we want to emphasize that the schedule is not partial, we will say that it is *complete*. If each activity is performed without interruption, we say that the schedule is *non-preemptive*. If this is not the case, we have a *preemptive* schedule. Preemptive schedules are not very common in practice. The interest in them is more due to the fact that some more basic scheduling problems are NP-hard when preemption is not allowed, but polynomially solvable otherwise. The preemptive solution is then often used for calculating lower bounds (minimization case) as a part of solution method for the non-preemptive original problem. For our application, preemptive schedules are not of much interest.

We call a schedule *left-justified* (or *semi-active*) if no activity (operation) can start earlier without changing the processing order on the resources. For most objective functions used in practice, it is sufficient to consider only left-justified schedules when optimizing. In this way, we remove the continuous part of the scheduling problem and we are left only with the combinatorial part (recall that we did just that in the preliminary model presented in Section 1.4). A schedule is called *active* if no activity (operation) can start earlier without delaying another. From this definition, we see that all active schedules are left-justified, but that the opposite is not necessarily true. One can show that for most objective functions, at least one optimal schedule is active.

We say that a schedule is *non-delay* if no resource is kept waiting when there is an activity (operation) available for processing. It can be shown that all non-delay schedules are active but that the opposite is not true. In a non-delay schedule we remove the possibility to delay a product in favour of another one that will be available in the near future. For non-preemptive problems like the ones we are interested in here, the restriction to only non-delay schedules can remove all optimal schedules. In the next section, we will show an example where this happens. We note here that limiting the search for schedules to non-delay ones is not necessarily a bad thing. It can be a good heuristic when the computing time available is short, since most non-delay schedules are fairly good.

3.4 A survey of common deterministic scheduling problems

In this section we will present some well-studied scheduling problems, in increasing order of modelling complexity. We will arrive at the so called *job shop scheduling* problem and its more advanced versions. These problems will serve as the basis for discussions in the next chapter when we present different solution strategies.

3.4.1 Single machine scheduling problems

The simplest scheduling problems include only one resource and a set of activities with given processing times to be performed under some constraints. Although single machine scheduling problems are of little direct practical interest, they have received lot of attention in the research society. This is partly due to the fact that they are simple enough to allow for thorough theoretical analyses, but more importantly that some problem versions arise as subproblems in solution methods for multi-machine scheduling problems. As we move on to even more advanced problems, the usefulness of solution methods for single machine problems decreases. Therefore we will not be very interested in them, and we will only mention a few examples.

For some problem versions, an optimal solution is found by simply sorting the activities according to some criterion. One example is if all activities are available at the start time and we want to minimize a weighted sum of the completion times of the activities. An optimal schedule is obtained if we sort the activities in decreasing order of weight over processing time. Other problems require more advanced solution algorithms but are still polynomially solvable. But even in this simple case there are many NP-hard problem versions. A well known-example is if we want to minimize the makespan when we have sequence-dependent setup times at the machine. This problem can be shown to be equivalent to the famous Travelling Salesman Problem (TSP) in combinatorial optimization. Single machine problems of interest for computing lower bounds when solving the job shop scheduling problem (defined below) are the minimization of makespan when release dates and deadlines are specified for the activities, and the minimization of maximum lateness with release dates and due dates specified. These problems are NP-hard, but fairly large instances can be solved to optimality quickly. The preemptive versions of these problems are polynomially solvable, and the solutions to these are sometimes used instead of the non-preemptive one for the calculation of lower bounds.

3.4.2 Parallel machine scheduling problems

A natural generalization of single machine problems is to have a number of parallel machines that can perform the activities. Neither these problems are very useful for our application, so we will only briefly discuss the perhaps most well-known problem variant. In this case all machines are identical and processing times are given for the activities, and we want to minimize the makespan. This is equivalent to balancing the load on the machines as much as possible. The problem is NP-hard, but a simple good heuristic exists for the problem: sort the activities in descending order of processing time and assign them in order to the first machine that becomes available. An interesting fact is that if the processing times are stochastic and follow a basic probability distribution, this simple algorithm (we modify it to sort according to expected processing time) is optimal when one wants to minimize the expectation of the makespan.

3.4.3 Shop scheduling problems

In the so called *shop scheduling problems* we have a set of m different machines and a set of activities with precedence constraints between them given. For each activity a unique machine and a processing time are specified. The most commonly studied objective is the minimization of makespan. If we impose no further restrictions on the problem, it is called *general shop scheduling* or *disjunctive scheduling*. The latter term comes from the so called *disjunctive constraints* that say that activities that are to be performed on the same machine cannot overlap in time.

Usually, the activities are grouped into *jobs* corresponding (in our application area) to the manufacturing steps of a product, in which case the activities are referred to as *operations*. The operations within a job cannot overlap in time. If there are no precedence relations between the operations in the jobs, the problem is called *open shop scheduling*. If the order of operations within each job is uniquely specified (i.e. the precedence constraints form a "chain" of operations for each job), we have the *job shop scheduling problem*, which like most other shop problems is NP-hard. If we have neither of these cases, we talk of a *mixed shop*. Job shop scheduling is perhaps the most studied problem in the scheduling community, and the literature on the subject is extensive. We will later give some references when we study specific solution methods for this problem, but we will already here mention Pinson (1995), which has a good general treatment of it. Other good general sources are the books mentioned in Section 3.3.1.

Before we look at some special cases of the job shop, we will give a simple example. We have three machines and two jobs in the job shop, and the objective is to minimize makespan. Data for the operations are given in Table 3.1. This example also illustrates that looking at only non-delay schedules is not sufficient for finding the optimum. In Figure 3.1, the optimum schedule is shown together with the best (and in fact the only) non-delay schedule.

A further restriction that can be made is that there is exactly one operation for each machine in all jobs[2], i.e. the number of operations is m. If in addition all jobs have the same processing order on the machines, we have the special case of *flow shop scheduling*. This problem can be specialized further to the so called *permutation flow shop*, where each machine have to process the jobs in the same order. This problem

[2]We note that many authors include this restriction in their definition of the job shop scheduling problem.

Job	Op. 1	Op. 2	Op. 3
1	M1/1	M2/3	M3/5
2	M2/3	M1/2	M3/1

Table 3.1: Example job shop. The operations of the jobs are to be processed in numerical order. Machines and processing time are specified for each operation.

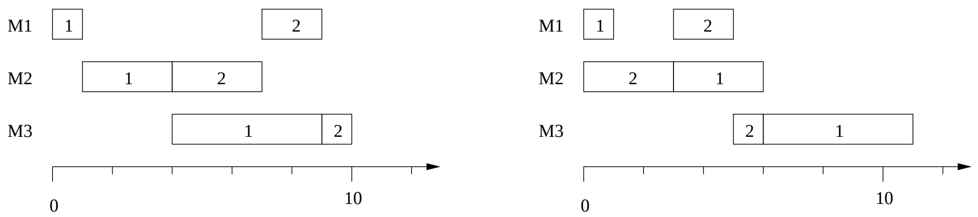

Figure 3.1: Left: Optimal schedule for the example job shop. Right: Best non-delay schedule.

is much simpler than the general flow shop, since one only has to find a permutation of jobs which implicitly decides the starting times of operations. The problem is still NP-hard, however.

3.4.4 Limitations of the job shop scheduling model

As we will see here, the job shop scheduling model is far too simple to be used directly in real-world scheduling applications like ours. However, the more advanced problems we are interested in have a job shop character to a large extent, so it is still an interesting problem for us. The main reasons why the basic job shop scheduling model is not suitable for a typical manufacturing environment are:

1. We cannot model the situation where an operation requires more than one resource.
2. We are not able to handle volumetric resources, e.g. parallel machines.
3. The model does not allow for alternative ways of performing activities.
4. It is assumed that there is unlimited buffer space for storing products between operations.
5. Transportation of products are not explicitly considered (in fact, it is assumed to be instantaneous). To some extent transportation times can be modelled by either including them in the operation processing times (assuming unlimited transport capacity) or by adding special transport operations done on "machines" that correspond to transport resources. However, this is not sufficient in a more general setting with e.g. limited buffer space and flexible routing.
6. A product lot size greater than one has to be modelled using one job for each item. This does not allow for cases where we do not want to distinguish between different items of the same product type.

7. Different release dates for products have to be handled artificially, by adding fictitious operations and machines.

8. We cannot handle the case where resources are unavailable during certain time periods.

9. We cannot model sequence-dependent setup times for machines.

10. The makespan is not the relevant objective in many real-world cases.

As can be seen, we have to overcome most of these restrictions in order to meet the requirements of our application (see Section 1.3.3). And even if we do so, there are still some modelling features we cannot handle: different time usage for resources needed on the same activity, resources needed for several consecutive activities, general routing flexibility, assembly etc.

3.4.5 More advanced models

For more advanced scheduling problems, not many generally accepted models exist. We will present a few cases here with fairly well-established names.

Flow shop with blocking

When there are no buffer space for storing products between machines, a product has to occupy a machine until its next operation is started. This is sometimes referred to as *blocking*. Flow shops with blocking have been quite extensively studied. For job shops, problems with blocking are more difficult since the occurrence of deadlocks is possible. This problem has only recently been studied, and we will review some work in the next chapter.

Flexible shop scheduling

If we allow parallel machines at each stage in a shop model, we get the *flexible* open/job/flow shop scheduling problems. Of these, the flexible flow shop problem has received most attention.

Resource constrained (project) scheduling problems

In the previous chapter we saw an example of a simple project scheduling problem: the central scheduling problem. A more advanced problem is the *resource-constrained project scheduling problem* (RCPSP), which has been extensively studied. It is a generalisation of the general shop scheduling problem with discrete time. Each resource (type) has a capacity and the amounts of different resources needed is specified for each activity. For each discrete time period we have the constraint that the usage of the resources cannot exceed the capacity. We have thus removed restrictions 1 and 2 in the list presented above. In the *generalized resource-constrained project scheduling problem* (GRCPSP) further extensions are made. Release dates and due dates can be specified for activities, resource capacity may vary over the time intervals, and precedence relations are generalized to be either start-start, finish-finish, finish-start or start-finish. These problems are presented in e.g. Herroelen and Demeulemeester (1995), and they can of course also be used in a production scheduling setting.

Chapter 4

Existing scheduling methods

In this chapter we will present methods from different research areas that have been used to solve scheduling problems with applications in manufacturing. We note that it is difficult to make a clear division of methods into different classes, since many scheduling techniques include elements from several research fields. Our way of organizing methods is just one possible.

For many of the techniques, we will first describe how they are used when solving the standard job shop scheduling problem. Then we will review the literature on solving more advanced problems with the technique, and discuss the possibilities to apply the technique on our application. Solution methods are not described in full detail, instead references for further reading are given. We note however that some of the method principles only briefly explained here will be treated in more detail in Chapter 6, where they are applied to our particular scheduling model.

4.1 Methods from Operations Research

In *Operations Research (OR)*, technical and economic decision problems are modelled and solved with mathematical methods. The research field is a part of the larger field of *Management Science*. Being an application-oriented area, OR overlaps a variety of more method-oriented areas such as statistics, probability theory and optimization. We are mainly interested in the latter aspect of OR here.

4.1.1 Mathematical Programming approaches

The aim of the mathematical programming approach is to state the optimization problem in an explicit form with an objective function and a set of constraints, using arithmetic expressions involving a number of continuous and discrete decision variables. Preferably, the expressions used should be linear in the decision variables, in which case strong theoretical results and analytical tools are available. The restriction to linearity is most important for the constraints, which means that they should be expressed as linear equalities or inequalities. From a problem modelling point of view this is a serious limitation, since many real-world constraints are impossible or at least difficult to express in this way. There are classes of constraints (e.g. logical conditions) that can be transformed into linear ones by introducing additional decision variables, but at the expense of an increase in the problem size which may be prohibitive for the solution methods that are to be used.

For readers unfamiliar with some of the terms used in the remainder of this section (branch-and-bound, cutting planes, Lagrangian relaxation etc.) we refer to any general text on discrete optimization, e.g. the book by Nemhauser and Wolsey (1988).

The basic job shop scheduling problem

We will first show a classic *mixed integer programming* (MIP) formulation[1] of the job shop scheduling problem. We have m machines and n jobs to process. Job i has n_i operations that have to be processed without preemption in the order of ascending index. Let R_{ij} denote the index of the machine that is to be used for operation j of job i, and p_{ij} the corresponding processing time. As our decision variables we will choose the starting times of the operations: let x_{ij} denote the starting time of operation j of job i. We will set the time scale such that the earliest starting time of any operation is 0. As objective we will choose the minimization of makespan, denoted by $C_{\max}$. Many other objectives, like e.g. mean flowtime, can easily be used in the same framework.

The constraints that determine the order of the operations within a job and assure that the processing of these operations cannot overlap in time are called the *conjunctive constraints.* With the aid of our notation given above, we can easily formulate the conjunctive constraints. We must have

$$x_{i,j+1} - x_{ij} \geq p_{ij},$$

for all $i = 1, \ldots, n$ and $j = 1, \ldots, n_i - 1$. The constraints that two operations that are to be processed on the same machine cannot overlap in time are called the *disjunctive constraints.* These are more difficult to express mathematically, we have that

$$x_{ij} - x_{kl} \geq p_{kl} \qquad \text{or} \qquad x_{kl} - x_{ij} \geq p_{ij},$$

for all $i = 1, \ldots, n-1$; $j = 1, \ldots, n_i$; $k = i+1, \ldots, n$; $l = 1, \ldots, n_k$ such that $R_{ij} = R_{kl}$. Note that we do not need such constraints between operations belonging to the same job, since this is already taken care of by the conjunctive constraints. The problem here is that we have a logical condition in the disjunctive constraints, but as mentioned earlier we only want simple linear inequalities. This problem can be overcome by introducing a binary decision variable for each pair of operations that are to be processed on the same machine (but not belonging to the same job): let y_{ijkl} take the value 1 if operation j of job i is processed before operation l of job k on the machine in question, and 0 otherwise. We will also need a large constant M. Here is now the complete problem formulation, where we for notational convenience introduce the set $A = \{(i,j,k,l) : i = 1, \ldots, n-1;\ j = 1, \ldots, n_i;\ k = i+1, \ldots, n;\ l = 1, \ldots, n_k; R_{ij} = R_{kl}\}$ for the disjunctive constraints.

[1]A subclass of mathematical programming which may include both real-valued and integer decision variables, with the restriction that the objective function and the constraints are linear.

$$
\begin{aligned}
&\min C_{\max} \\
&\text{s.t.} \left\{ \begin{array}{lll}
x_{i,j+1} - x_{ij} \geq p_{ij} & i = 1, \ldots, n; \; j = 1, \ldots, n_i - 1 \\
C_{\max} - x_{in_i} \geq p_{in_i} & i = 1, \ldots, n \\
x_{kl} - x_{ij} + M(1 - y_{ijkl}) \geq p_{ij} & (i,j,k,l) \in A \\
x_{ij} - x_{kl} + M y_{ijkl} \geq p_{kl} & (i,j,k,l) \in A \\
x_{ij} \geq 0 & i = 1, \ldots, n; \; j = 1, \ldots, n_i \\
y_{ijkl} \in \{0,1\} & (i,j,k,l) \in A
\end{array} \right.
\end{aligned}
\tag{4.1}
$$

This formulation clearly shows both the discrete nature (ordering operations on the machines) and the continuous nature (assigning start times for the operations) of the job shop scheduling problem. The instances of this problem can in principle be attacked with a standard MIP solver (that typically uses the exact branch-and-bound technique), thus removing the need for developing a special purpose algorithm. However, only problems of very modest size can be solved in this way because of the combinatorial explosion associated with the disjunctive constraints and the corresponding binary variables. To the best of our knowledge, no general-purpose approximate MIP solvers using fast heuristic methods exist that could be used instead of an exact solver. Neither has (4.1) shown to be a good starting point of specialized (primal) heuristic approaches. Applegate and Cook (1991) used it as a starting point for a *cutting plane* algorithm calculating lower bounds (i.e. it can be regarded as a dual heuristic). The results obtained were only slightly better than the standard bounds calculated by solving a set of the simplified problems, but required considerable computational effort.

Deadlock-free scheduling and material handling

As mentioned earlier, limited bufferspace for parts restricts the scheduling possibilities of activities in a manufacturing system. A situation that may occur is a *deadlock*, where parts cannot move anymore because they are in a circular wait state. In a manned system such situations can be resolved with some extra effort, but an automated system do not necessarily have such an opportunity. In the latter case, it is important that the control system always generates deadlock-free schedules.

Ramaswamy and Joshi (1996) extend a MIP formulation similar to (4.1) (but with mean flow-time as objective instead of makespan) to include the case with no buffers between machines, and proves that this formulation provides an optimal deadlock-free schedule. In order to be usable in an automated system, the scheduler must also take the transport of parts (typically performed by a robot) between machines into account. This can be done by modelling the transporter as a resource just as the machines, and by extending each job with transport operations. But this more than doubles the number of operations in each job, leading to a very large increase in disjunctive constraints. The authors show that this leads to a significant solution time (over a minute) already for a very small problem with 3 machines, 1 robot and 4 jobs. Instead they propose a heuristic modification of the optimal deadlock-free schedule without transports, that adds the material handling tasks afterwards. This method performs reasonably well when the transport times are small compared to processing times.

Other extensions

Mathematical programming formulations of an entirely different type than (4.1) have been proposed in the literature, not with the aim of being suitable for a standard MIP or IP solver but rather to provide a basis for a heuristic approach. Discrete time with a given time horizon is used, and the time unit used for specifying e.g. operation times in problem instances is taken as large as possible in order to keep the complexity of the model down (at the expense of modelling accuracy). Machines are grouped into machine types with the capacity available (i.e. the number of machines of the type in question) specified at each time unit. Operations in jobs can have quite general precedence constraints, and required timeouts between successive operations can be specified. For each operation the possible machine types and the corresponding processing times are specified. Apart from the deteriorated time resolution, this is a much more advanced model than the standard job shop scheduling problem.

This model is first presented in Hoitomt et al. (1993). The primary decision variables they use are operation starting time and the machine type selected for the operation. Instead of disjunctive constraints, machine capacity constraints are used together with additional binary variables, one for each combination of operation, machine type and time unit. Although these variables can be of large numbers, the problem formulation has some nice properties that makes it usable. The authors use a heuristic based on *Lagrangian relaxation* to produce high-quality solutions and good lower bounds. The time required to produce a solution for some real-world example problems with some 30 machines and a total of a few hundred operations is a couple of minutes per problem, which must be regarded as very reasonable. In Wang et al. (1997), the Lagrangian relaxation solution methodology is improved giving better lower bounds and faster execution times compared to those of earlier attempts, but for a slightly simpler model (precedence constraints are of the standard job shop type and no required timeouts can be specified).

In Luh et al. (1998a) further extensions of the above model are made in order to include group-dependent setups, finite buffers and a more general form of flexible routing. The model can handle a special case of the situation when setup times are not independent of the sequence of operations on machines. Operations with similar setup requirements are forced to be processed in groups on a machine, with the setup time for the entire group specified. Note that this is not the general case of sequence-dependent setups, since the group setup time is sequence independent. A finite buffer is modelled as a machine (type) with a certain capacity, with processing times for buffering operations regarded as unknown variables. The model also includes alternative routes (of fork-join type) for products, apart from the flexibility in choosing machine type for a single operation. The authors also show how to approximately handle long time horizons by grouping time units into so-called *enumeration steps* (if the time resolution is one hour, an enumeration step is typically one working day) and considering only the total capacity within each such enumeration step. The objective function used weighs on-time deliveries and low in-process inventory together. Numerical results are presented for scheduling problems from a switchgear manufacturing facility arranged as a flow shop.

These recent approaches are very impressive compared to earlier mathematical programming attempts, both with regards to modelling power and solution efficiency. One big advantage is that solution quality is quantifiable, since the methods automatically give lower bounds. The solution algorithms are quite complicated however, being composed of many complex sub-algorithms that are not easily extended with new types of constraints. The methods still lack the capability to handle some of the vital model features we are interested in, e.g. multiple resource usage and product lot sizes. Also, the solution times presented in the literature are generally too long for our application. For these reasons, we will not at this stage attempt a mathematical programming approach for the scheduling system, but we suggest that the development in the area is closely followed.

4.1.2 Neural network methods

Neural networks (see e.g. the book by Hertz et al. (1991)) is normally regarded as a branch of Artificial Intelligence rather than of Operations Research. We still prefer to present the subject in this section, since when neural networks are applied to discrete optimization problems, one uses much of the mathematical programming theory. As is the case for many other discrete optimization problems, most neural network approaches made for the job shop scheduling problem are not competitive with other solution techniques. Recently, Lagrangian relaxation and neural network techniques have been combined giving significantly improved performance. In Luh et al. (1998b) the authors use this method based on the model from Wang et al. (1997) with good results. But since neural network methods share the earlier mentioned limitations of the mathematical programming approach, they are not interesting for our scheduling system, and we will not go further into details of this research field. For readers still interested in the subject, references can be found in Luh et al. (1998b).

4.1.3 Methods based on the disjunctive graph model

A *disjunctive graph*[2] is a suitable tool for the modelling and solution of the general shop scheduling problem (and its more restrictive versions) with makespan objective. The nodes in the graph represent the activities (operations) and are labelled with the corresponding processing times. There are in addition two dummy nodes, the *source* and the *sink*. There are two sets of arcs in the graph. The first is the directed *conjunctive arcs* that represent the precedence constraints between activities. There is a conjunctive arc from a node A to a node B if there is a constraint stating that the activity corresponding to A must end before the one corresponding to B is started. In addition, there is a conjunctive arc from the source to each node without predecessor, and one from each node without successor to the sink. Note that with the elements we have added so far in the graph, it is equivalent to the *potential task graph* discussed in Section 2.4.1, the only exception being that the times are here associated with nodes instead of with arcs.

The second set is the undirected *disjunctive arcs* which represent the disjunctive constraints. In the general shop scheduling problem there is a disjunctive arc between two nodes if and only if the corresponding activities use the same machine. In job shop scheduling one needs only disjunctive arcs between activities in different jobs, since the conjunctive constraints force two operations in the same job not to overlap regardless if they use the same machine or not. In the mixed shop case, one needs a disjunctive arc between nodes corresponding to operations in the same job but whose order is not determined by the precedence constraints. We will illustrate the disjunctive graph model with a simple job shop scheduling example[3], having two

[2]A perhaps more proper name is *conjunctive-disjunctive graph*, but we (and most other authors) prefer to use the more convenient shorter version.

[3]It is in fact a flow shop, but that is not important here.

machines and two jobs. The data for the operations are given in Table 4.1, and the disjunctive graph for the problem is shown in Figure 4.1.

Job	Op. 1	Op. 2
1	M1/3	M2/4
2	M1/5	M2/6

Table 4.1: Example job shop. The operations of the jobs are to be processed in numerical order. Machines and processing time are specified for each operation.

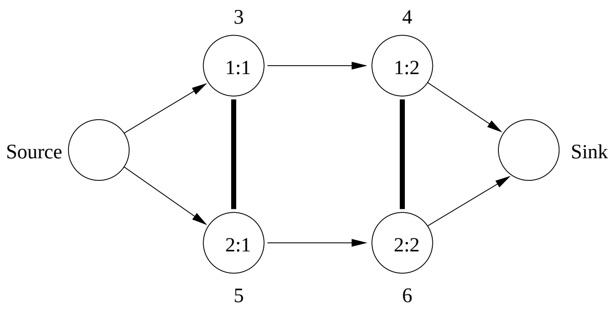

Figure 4.1: A disjunctive graph for a simple job shop scheduling problem. The job and operation number is shown inside the node circles and the corresponding processing time is given next to the node. The disjunctive arcs are shown with bold lines for clarity.

The disjunctive graph is to be seen as a basis for solution methods, and not as a graphical presentation tool. Graphs for small problems can be used for illustrating principles, but graphs for larger problem are impossible to view graphically since they are cluttered with disjunctive arcs.

Disjunctive graph theory

We will now present some properties of disjunctive arcs that are important for solution methods. Such methods work by selecting an orientation for each disjunctive arc (i.e. making it directed instead of undirected), thus fixing the order between the corresponding activities. A set of disjunctive arcs for which the direction has been fixed is called a *selection* (or *orientation*). If all disjunctive arcs have been fixed we have a *complete selection*, otherwise it is called a *partial selection*. A complete selection is called *feasible* if the resulting directed graph is acyclic[4]. With the length of path between the source and the sink in the directed graph defined by a feasible selection we mean the sum of the processing times associated to the nodes in the path. Such a path in the graph with largest possible length (there can in general be more than one) is called *critical*. We now have two important results:

- There is a one-to-one correspondence between the set of feasible selections and the set of feasible left-justified schedules.

[4]We note that some authors include this in their definition of a complete selection.

- The length of a critical path in the directed graph resulting from a feasible selection is equal to the makespan of the corresponding left-justified schedule.

This means that one never has to work explicitly with starting times of activities in a disjunctive graph based solution method, instead one concentrates only on orders between activities. When the solution process terminates with the best feasible selection found, it is straightforward to derive the starting times. Another fact that makes disjunctive graph based methods powerful is that one can use well-known methods from graph theory for finding longest paths and detecting cycles in a fast way.

Branch-And-Bound

A lot of effort has been put into developing exact methods based on the disjunctive graph model for the job shop scheduling problem. Today extremely advanced branch-and-bound methods exist that can solve problems with up to about 15 machines and 15 jobs. In most methods, a node in the search tree is specified by a selection in the disjunctive graph. When a search node is selected for branching, the most common approach is to select a disjunctive arc not in the selection and then create two child nodes, one for each possible direction of the arc. In Figure 4.2, a search tree for the above job shop example is shown to illustrate this.

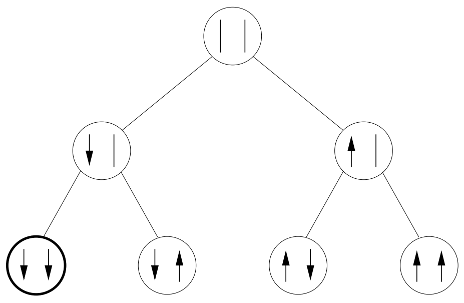

Figure 4.2: One possible search tree for the job shop scheduling problem modelled with the disjunctive graph of Figure 4.1. The selection of disjunctive arcs is shown inside the search nodes. The search node corresponding to the optimal solution is drawn with a bold circle.

In order to avoid explicitly exploring a large portion of search tree, it is important that the lower bound calculated at each search node is as good as possible. A simple and fast way of doing this is just to remove all disjunctive arcs not in the current selection and to calculate the longest path in the remaining directed graph. A better lower bound (but computationally more expensive) can be calculated by discarding all non-fixed disjunctive arcs for all but one machine. This leads to a single machine problem which is usually possible to solve quickly despite the fact that it is NP-hard. We can of course do this for all machines and then pick the best bound obtained. Several other ways of calculating lower bounds exist.

Apart from lower bound calculations, many other issues are important, e.g. which non-fixed disjunctive arc to select next for branching. Many of the already mentioned references treat branch-and-bound methods for the disjunctive graph model in more detail, e.g. Pinson (1995), Błażewicz et al. (1996) and Brucker (1998).

The shifting-bottleneck heuristic

When larger problems are to be solved, exact methods cannot be used any more. One of the more well known approximate methods for the job shop scheduling problem is the *shifting bottleneck heuristic*, originally proposed by Adams et al. (1988). It is based on the same idea as was described for calculating lower bounds for use in branch-and-bound methods. The algorithm goes through a number of stages equal to the number of machines in the problem. At the beginning of stage number k, the operation order on $k-1$ machines has been determined (i.e. all disjunctive arcs for these machines have been fixed). Then a single machine problem is solved for each of the remaining machines. The machine which causes the largest increase in makespan is identified as the "bottleneck". Since it seems reasonable to schedule such a machine before others, this machine is included in the set of scheduled machines (i.e. the disjunctive arcs corresponding to the machine is fixed according to the solution of its one-machine problem). Finally, at the end of the stage, the algorithm makes a local re-optimization of the schedule for the k so far selected machines.

The book by Pinedo (1995) has a good example illustrating the steps in the method. Many variants of the algorithm have been proposed, e.g. in Applegate and Cook (1991). Vaessens et al. (1996) presents computational results for many shifting bottleneck heuristic versions as well as for other heuristics.

Meta-heuristics

A *meta-heuristic* is a general algorithmic principle that can be used as a basis for constructing approximate methods for a variety of problems. The most well-known meta-heuristics are *simulated annealing* (see e.g. Aarts and Korst (1989)), *tabu search* (see e.g. Glover and Laguna (1997)) and *genetic algorithms* (see e.g. Goldberg (1989)). The former two are based on the principles of *local search*, which is to iteratively move from a (complete) solution to another *neighboring* solution by making some small changes. The disjunctive graph model is very suitable for local search. The simplest way of moving from one solution (represented by a complete selection) to another is just to reverse the direction of a disjunctive arc. It is common to restrict changes to only arcs lying on critical paths. It has been shown that it is possible to reach an optimal solution in this way regardless of the starting solution.

Especially tabu search approaches based on the disjunctive graph model have been successful. The algorithm by Nowicki and Smutnicki (1996) is one of the most efficient methods known for job shop scheduling. Barnes et al. (1995) present tabu search approaches for job shop scheduling as well as for other scheduling problems. Methods based on simulated annealing can also give quite good results, but require rather long running times. Genetic algorithms are not that suitable for the disjunctive graph model, and the approaches suggested so far are not competitive with the best methods. The computational study by Vaessens et al. (1996) includes many methods based on the meta-heuristics described here.

Extensions of the basic model

When the disjunctive graph model can be used, very efficient solution methods can be constructed. Unfortunately, the model is not very easy to extend to more general cases. Features like limited buffers seem very difficult to model with a disjunctive graph. Also, the disjunctive graph loses much of its strength when an objective other than makespan is considered, since there no longer is a connection between the objective value and the length of critical paths. The most advanced model based on disjunctive graph ideas that we are aware of is presented in Verhoeven (1998). The problem treated is referred to as the *resource-constrained scheduling problem (RCSP)* and it is similar to the RCPSP problem described in Section 3.4.5.

In this problem, each resource can have an integer capacity. For each activity, a number of possible resource sets for performing it can be specified together with processing time (i.e. multiple resource usage and a limited form of routing flexibility is possible). Activities can have general precedence constraints between them. The objective is to minimize makespan. A solution of the problem cannot be described in a single disjunctive graph, instead it is defined by a resource assignment for the activities and a selection in a graph that depends on the resource assignment. A tabu search algorithm is presented for the problem and it is tested on some special cases of the RCSP.

Since there is still a long way to go from the RCSP to a model which meets our requirements, we doubt that an approach based on a disjunctive graph is suitable for our application.

4.2 Methods from Artificial Intelligence

Artificial Intelligence (AI) is a huge research field, for a general treatment see e.g. Winston (1984) or Russell and Norvig (1995). Some parts of this field are of interest in combinatorial optimization in general and scheduling in particular. Compared with Operations Research, the work on scheduling with AI methods has been more directly directed towards industrial applications. Collected articles discussing such applications as well as more theoretical issues can be found in Zweben and Fox (1994) and Brown and Scherer (1995). In this section, we will briefly discuss some AI methods and then see how they can be applied to scheduling problems of the kind that is of interest to us.

4.2.1 Introduction to AI methods

Artificial Intelligence has a large number of branches. One of them is *search*, which concerns methods for finding sequences of actions in order to achieve some kind of goal. Some of these methods will be of interest for us in discrete event models. Section 4.3 treats such models, and we will discuss relevant search methods there. A particular class of search methods that we will discuss here is devoted to so called *constraint satisfaction problems*.

Another major branch is *logic*, where one develops formal languages for encoding knowledge and methods for carrying out reasoning within the language. So called *first-order logic* where one can express relations between objects is perhaps the most widely studied formal logic language. The well-known logic programming language *Prolog* is based on first-order logic. Prolog and the like allow for programming at a very high level, but the price paid for generality is efficiency problems in the logical inference processes. A closely related area is *expert systems*, in which one attempts to store information given by human domain experts in *knowledge bases*, and use it for automatically making diagnoses or taking decisions. There have been some attempts with knowledge-based systems in production scheduling, but we have not found any of these approaches suitable for our application.

Constraint propagation techniques

Constraint satisfaction problems (CSP) are combinatorial problems in which the goal is to satisfy a set of constraints. Note the close connection to combinatorial optimization problems (COP), where one in addition wants to optimize some criterion while obeying the constraints of the problems. Although CSPs can be said to be a special case of COPs, it is considered a separate research area since the applications studied generally differ from the ones in combinatorial optimization. In COPs one usually studies subsets of constraints that are fairly easy to handle, whereas the research on CSPs is directed towards more general classes of constraints and problems where feasibility is difficult to achieve. This has lead to different types of solution techniques. The reason why we are interested in CSP techniques is that they can be useful in difficult COPs such as ours as a complement or alternative to classic COP techniques.

An important component in CSP techniques is *constraint propagation*. When a decision is taken (e.g. the assignment of a value to a variable) in the search process for the problem, the consequences of this is analyzed for the constraints, and the result can be that the set of possible values for other decision variables are narrowed down or that unavoidable violation of a constraint is discovered. This means that large parts of a search tree may be cut off without explicitly exploring it. A trade-off has to be made between how many search nodes to explore and the amount of constraint propagation done at each node. The propagation process can be a combination of general techniques and techniques specialized for certain types of constraints.

We will now only briefly describe some further CSP basics. For a more general treatment we refer to the literature, e.g. Kumar (1992). A *binary CSP* is a problem which can be described by a set of decision variables with a discrete set of possible values and constraints involving only pairs of variables. We note that many other types of CSPs can be transformed into binary CSPs. The problem can be represented by a *constraint graph*, where each node corresponds to a variable and there is an undirected arc between two nodes for each constraint between the two variables in question. An arc between nodes/variables i and j is said to be *consistent* in the direction from i to j if there for all possible values of i is at least one possible value of j such that the corresponding constraint is not violated. If the arc is not consistent, we can directly remove some value(s) from the set of possible ones for i. There exist several variants of algorithms for making the entire constraint graph arc consistent. Full or partial consistency checking can be made either before a search process starts or at each search node.

Constraint logic programming

In the mid 80's, research on using constraint propagation techniques in first order logic in order to improve the efficiency of logical reasoning methods were done. This development together with the introduction of richer data structures into logic programming languages led to the birth of *constraint logic programming (CLP)*. In an ordinary logic programming language the objects used can represent just about anything, but all relations between objects have to be explicitly stated in order to allow for logical reasoning. In CLP one also has objects that have a meaning in a special application domain (e.g. the arithmetic domain), with associated operations. This allows for implicit expression of relations between objects using more general constraints. CLP has not only richer semantics, but also improved efficiency due to the possibility to use specialized constraint solving techniques (e.g. the simplex method for linear constraints in real-valued variables). There is a whole family of CLP languages depending on the application domain(s) supported. A good introduction to CLP is given in Frühwirth et al. (1993).

4.2.2 Constraint-based approaches to job shop scheduling

Many of the modern branch-and-bound methods based on the disjunctive graph model for job shop scheduling (see Section 4.1.3) use some elements of constraint propagation. This is usually done by calculating a time window for each activity within which it has to be performed. When a pair of activities is ordered, propagation of this decision may lead to some time windows being narrowed and sometimes inconsistency can be detected if an activity no longer fits in its time window. The disjunctive graph functions in this case as a constraint graph.

Caseau and Laburthe (1995) use constraint propagation to a large extent with a slightly different approach called *task intervals.* A task interval is actually a set of activities to be performed on the same machine whose time windows fulfil some requirements. The constraint propagation techniques for task intervals are not only used within a branch-and-bound algorithm, but also for calculating lower bounds, within a greedy method and in a local search method. All these parts are then integrated into a hybrid algorithm which is very efficient.

4.2.3 Extensions to more advanced models

For more advanced scheduling problems, a CLP-approach seems to be the most promising alternative due to the flexible modelling capacity. ILOG SCHEDULE is a such a product, based on the generic CLP-tool ILOG SOLVER and specialized for production scheduling. In Baptiste and Le Pape (1995), the constraints that can be modelled with SCHEDULE is presented. Baptiste et al. (1995) describe the propagation mechanisms used in more detail and report some computational results. Some of the modelling features possible in SCHEDULE are:

- Many types of temporal constraints. This includes general precedence constraints (start-start, finish-finish, finish-start or start-finish) and minimum and maximum delays between activities.
- Several types of resources. One can have unary resources, volumetric resources and state resources. The latter can be used when an activity can use a resource only under specific conditions.
- Resource utilization constraints. The required resources can be specified for activities. It is also possible for an activity to consume or produce resources.
- Alternative resources for an activity (i.e. a limited form of routing flexibility) and optional activities.
- Transition times between activities. This can be used to model sequence-dependent setup times.

This approach is certainly a candidate for our scheduling system, since the versatility of the CLP language should make it possible to model even more cases. There are two question marks however. The first is how easy it is to include the remaining modelling requirements of our application (limited buffers, multiple resources for an activity with possibly unequal time usage, product lot sizes, total costs as objective instead of makespan etc.). The other is how well the solver will do given only a short computing time as in our application, especially if there are many constraints which cannot be handled by the specialized propagation methods.

4.3 Discrete event models

In Section 1.4, we introduced the concept of a *discrete event dynamic system* and argued that it is one of the most natural and straightforward ways of describing a scheduling problem in a manufacturing system. It is also very commonly used in practice, especially in industry. The power of discrete event approaches lies mainly in its modelling capacities. It is fairly easy to customize a model to accurately describe the dynamic behaviour of a particular manufacturing system. This also makes it suitable for other purposes than scheduling, e.g. the surveillance and control of a system. Discrete event models are also very suitable for describing dynamic and stochastic scheduling problems.

4.3.1 General solution principles

A drawback of a discrete event model is that one has to resort to only one way of scheduling it: from front to back in time. In e.g. a disjunctive graph model there is much more freedom in which order to make scheduling decisions, and it is easy to move from one solution to another. If we for a discrete event scheduling problem have a solution in form of an action sequence and change a decision somewhere in the middle of the sequence, we have to reschedule the entire last part of the sequence. This means that local search methods are not easy to apply to a discrete event model. Instead we are restricted to mainly three solution method principles (of which the first can be said to be a special case of the second):

- Construct a single action sequence, making each decision according to some more or less advanced rule. This is mainly interesting in a real-time application or for making repeated simulations in a dynamic/stochastic scheduling problem, but neither of these cases are of interest to us.
- Construct a series of action sequences until the available computing time is used up and then return the best found. The latest generated sequence may or may not depend on previous sequences, and randomization is usually used in some way.
- Systematically organize all possible action sequences in a decision tree and use some tree search method from Operations Research or Artificial Intelligence.

A discrete event model has the advantage that since it describes the actual behaviour of the modelled system (i.e. all constraints of the problem are explicitly handled), every action sequence that leads to the goal state is a feasible solution. This means that it in most cases is easy to find one or a few feasible solutions extremely quickly[5]. But the approach also has a major disadvantage, which we now will explain.

Although the search tree in a discrete event model is restricted to only feasible solutions, it is much larger than for e.g. a corresponding disjunctive graph model.

[5]In many methods mainly based on other models, a discrete event model is often used to construct an initial solution.

This is due to the fact that we often have decisions that are independent of each other (it may be possible to take several decisions at the same time point for example). We will then have many actions sequences that are totally equivalent to each other, and we can have several search nodes corresponding to the same system state. In Figure 4.3, this is illustrated for the simple job shop problem presented in Section 4.1.3. Compare this search tree with the disjunctive graph search tree in Figure 4.2.

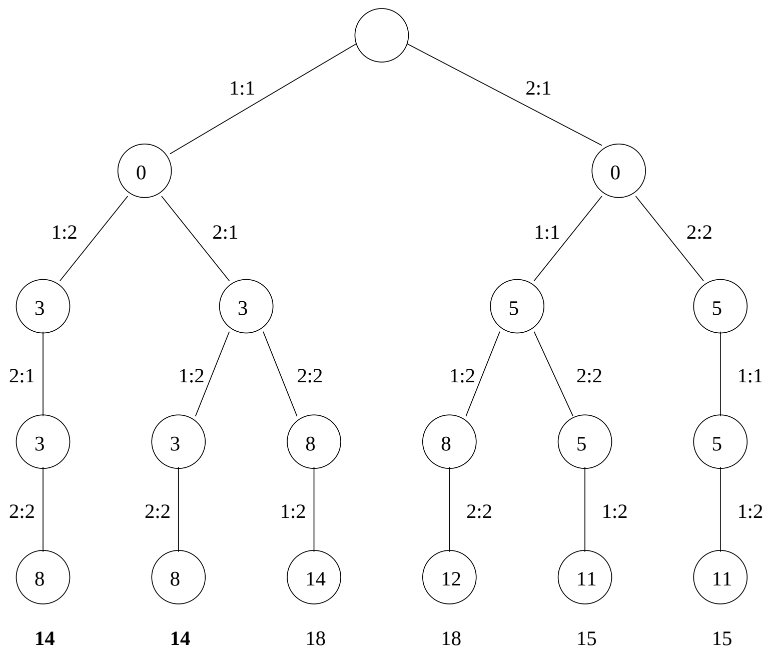

Figure 4.3: A decision tree for the job shop example specified in Table 4.1, described with a discrete event model. We define an action to be the starting of an operation. The action taken to reach a state represented by a node is shown at the arc leading to the node. The time at which the last decision was taken is shown inside each node, and the makespan of a complete schedule is shown below the leaf nodes. Optimal values are shown bold.

For a problem which also allows for e.g. a disjunctive graph model, it is in general much more difficult to find optimal or near-optimal solutions with a discrete event model. By checking for repeated states during a tree search process one can avoid exploring equivalent paths to some extent, but the number of search nodes that has to be explored in the tree may still be huge.

4.3.2 Methods based on priority rules

Priority rules for selecting the next activity to start in a scheduling process have been extensively studied over the years for different scheduling objectives, due to the fact that they are easy to use in real-world applications and that they also can be applied in dynamic/stochastic environments. Most work in the area has been devoted to cases without flexible routing, so that it is uniquely specified in which machine queue a product is put after its latest operation. In a given situation, a priority value is calculated for each activity that can be started next and the one with the highest priority is selected. If there is a tie between two or more activities, it can be broken by calculating another priority value or by simply selecting an activity randomly or according to some predefined order. Examples of some commonly used priority rules are (commonly used abbreviations are shown inside parentheses):

- First-In-First-Out (FIFO).
- Shortest processing time (SPT). The activity with the shortest processing time is chosen first.
- Shortest setup time (MINSEQ). This rule applies to the case where we have sequence-dependent setup times at a machine. A more advanced version (FIXSEQ) solves a single-machine scheduling problem (see Section 3.4.1) for the products that are currently in the machine queue and then selects the first product in this schedule.
- Most work remaining (MWKR). The activity for the job with the largest remaining processing time (including the activity in question) is chosen first.
- Earliest due date (DD). The activity belonging to the job with the earliest due date is selected. Used for due date related objectives.
- Least slack (SL). The activity belonging to the job with the smallest slack is chosen first. Also due date related.
- Work in next queue (WINQ). The activity whose successor are to be processed in the machine with smallest queue (with respect to processing time).

The priority rules can be categorized in various ways. One classification is if they take into account global information (like WINQ) or just information at the queue in question (the remaining examples). Another is if the priority value of an activity is time dependent (like FIFO, MINSEQ, SL and WINQ) or if it can be calculated once and for all (SPT, MWKR, DD). By combining several simple priority rules one can form more advanced versions. A survey of the subject can be found in Haupt (1989). A slightly different approach than standard scheduling rules is described in Holthaus and Ziegler (1997), where global information is taken into account.

Greedy methods

The well-known Giffler-Thompson algorithm (described in e.g. Błażewicz et al. (1996)) can be said to form the basis of most rule-based approaches that generates a single schedule for a (static) deterministic job shop scheduling problem. Since a decision taken is never changed, we can refer to it as a *greedy method*.

More advanced methods

If ties between activities are not too uncommon when priorities are compared, one can extend the Giffler-Thompson algorithm in a simple way by running it several times and breaking ties randomly. This might improve the "one-shot" value slightly. A more advanced (but also quite strange) approach is taken in Dorndorf and Pesch (1995). There a genetic algorithm (see e.g. Goldberg (1989)) controls the generation of new schedules. An individual in the population is defined by a string with $n-1$

entries, where n is the number of activities in the problem. Entry i in the string is a priority rule, and this rule is to be used at the i:th stage of the Giffler-Thompson algorithm. The algorithm requires rather long running times, and the results obtained are not competitive with disjunctive graph based heuristics.

Applicability to our problem

The strength of priority rules lies in the extremely quick execution time and the fact that they can be used in dynamic scheduling. We have a static case, and even if our computing times are short we want to use all time available in order to find a schedule that is as good as possible. In view of this, an approach purely based on priority rules is probably not very good in our case.

4.3.3 Methods based on Petri net models

As mentioned in Section 2.1, different types of Petri nets have been widely used for the modelling of manufacturing systems. An area that has received particular attention is that of *Flexible Manufacturing Systems* (FMS), perhaps due to the fact that the discrete event nature of a Petri net model is suitable for the control of such an automated system. Initially, the possibilities of scheduling were not addressed, instead simple rules like first-in-first-out (FIFO) were used to control the execution of the system. Later, both on-line and off-line scheduling problems for FMS have been studied. More recently, Petri net based methods have been recognized as a promising tool for a broader range of problems, including the scheduling of general manufacturing processes (not necessarily fully automated). Problems from other areas such as the scheduling of computer processes and the scheduling of restoration actions for power systems have also been attacked with Petri net methods, but we will not discuss these cases further since the models used are not applicable to our scheduling system.

Next, we will show how the standard job shop scheduling problem for minimizing makespan can be formulated in a place-timed Petri net framework, and how this model easily can be extended to cover much more realistic scheduling applications. Then we will present an overview of the research made in the area.

Modelling the standard job shop scheduling problem

The standard approach for the modelling of job shops uses two kinds of places: *product places* (also referred to as *process places*) and *resource places*. The product places represent the status of a product during the manufacturing process: being processed in a machine, waiting in a buffer etc. The product places corresponding to operations are timed places, with the time delay equal to the processing time of the operation in question. In the simplest case the product places form chains, one for each product, which represent the technological order of operations done on the product. Each product chain has a start place which corresponds to the product waiting for the first operation and a goal place which represents the product being completed. The transitions that link the product places in a chain represent events like the start or end of an operation. Different product chains are only connected through the resource places, which represent the availability of machines.

The initial marking consists (in the basic case) of one token for each product start place and one token in each resource place. The goal is to reach a marking where there is a token in each product goal place. The scheduling problem is thus to find a firing sequence which reaches the goal state in minimum time. In terms of the corresponding reachability graph, this is equivalent to finding the optimal path from the initial state to the goal state in the graph. Since each firing of a transition moves the system one step closer to the goal state, the number of firings required to reach the goal state is independent of the firing sequence. If we construct a tree search algorithm, we can take advantage of the fact that we know a priori the depth of the search tree.

An important issue is how we keep track of time and how we select the next transition to fire, given a current state of the system and the time elapsed to reach this state. In the general case, we allow any transition that is enabled without regarding time to fire next. If this transition has a timed input place (i.e. a product place corresponding to an operation), the time of the system is advanced to the moment when the token in the timed input place becomes available. A restriction of the firing possibilities that is often done is termed *eager firing*. We say that transitions are *eager-to-fire* when we follow the rule that they fire as soon as possible. The firing of a transition can of course still be delayed if a choice is made to fire another transition enabled at the same time which is in conflict with the former, but as soon as there are no such conflicting transitions it will fire. Another way of explaining the concept of eager firing is to say that "we only advance the time of the system when there are no more enabled transitions at the current time". As the reader may already have guessed, eager firing in Petri net scheduling is equivalent to the definition of *non-delay* scheduling made in Section 3.3.2.

We will now illustrate the issues discussed above with an example. In Figure 4.4, a Petri net model for the simple job shop example given in Section 4.1.3 is shown with an initial marking. Note that between the places corresponding to operations (these timed places have a number below them signifying the time delay) there is a place representing the product being in a buffer. These places correspond to the implicit assumption of unlimited buffers in the basic job shop model, and assure that the Petri net never ends up in a deadlock state (not counting the goal state of course).

In the figure, the reachability graph from the initial marking in the Petri net is also shown. The progress of the first job (J1) is shown on the horizontal axis and the progress of J2 is on the vertical axis. The upper left node is the initial marking and the lower right node is the goal marking. There is no unique time associated to a node, since this is a property of the path used to reach the node. The reachability graph clearly illustrates the point that there are many ways to reach a certain marking, some of which may be equivalent with respect to time. The impact of this on the performance of search methods was discussed in Section 4.3.1. The arcs lying on optimal paths are drawn with bold lines.

The reduction of the reachability graph due to the eager firing restriction is shown in the figure as well. In this case, the reduced graph still contains paths that are optimal in the original graph, but as the example in Section 3.4.3 showed, this is not always the case.

Extensions of the standard model

To model a situation with no buffer between two machines, we just remove the buffer place between the two operation places. In general, when buffers are finite, and in particular when there are no buffers, the possibility of deadlock in the system model must be taken into account. In some cases (e.g. in manned shops or in FMSs with

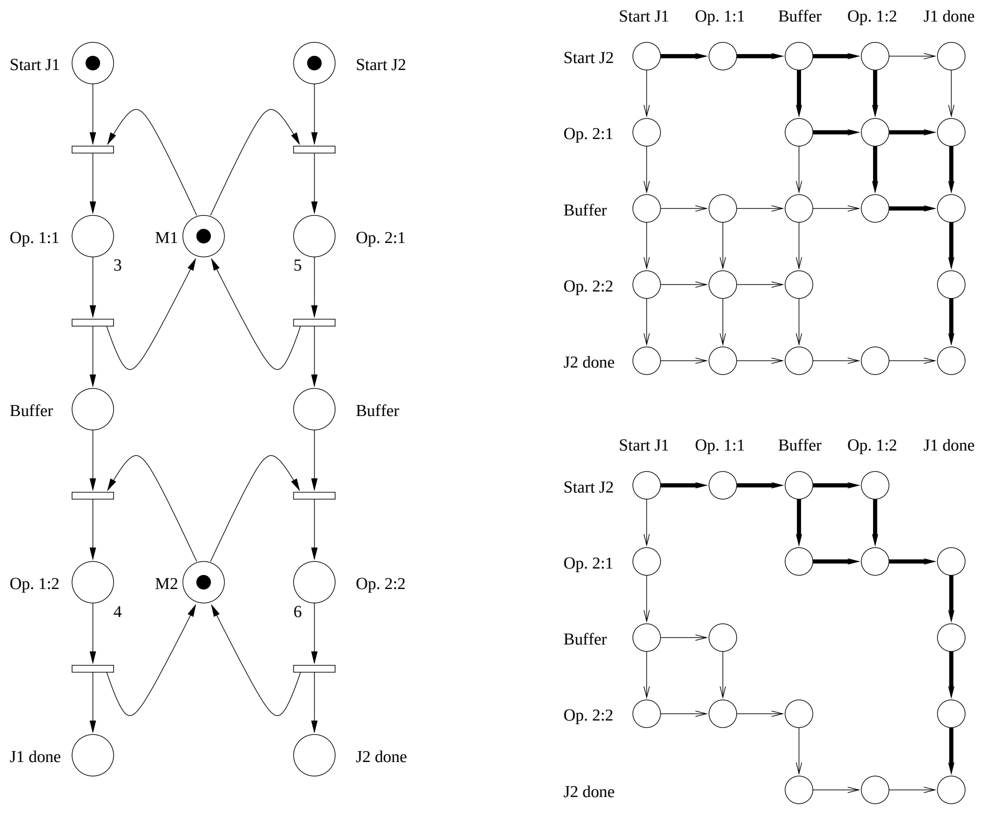

Figure 4.4: Left: Petri net model of the job shop scheduling example. Upper right: The reachability graph from the initial marking. Optimal paths are drawn with bold lines. Lower right: Reachability graph when transitions are eager-to-fire.

recovery possibilities), this situation can be handled in the actual system when it occurs. But this requires some extra effort and time, and it is of course better if the scheduling system never produces a deadlock situation. For a Petri net where deadlocks can occur, a search algorithm must therefore be able to find a way around such states. We note that in some cases with restricted buffering, the eager firing rule may cause all paths to the goal state to be cut off. Take e.g. the example above, remove the buffer places and also change the order of the operations of job 2 (M2 is used before M1). After starting the first operation of either job, the eager-to-fire transitions will then directly start the first operation of the other job, and the reader may check that this now is a deadlock state.

A more advanced model than the basic job shop scheduling problem is easily built in the Petri net framework. We have already discussed the case of no buffers, and in Figure 4.5 we show several other extensions. Case A shows that if several items of the same product are to be manufactured, this can be modelled in a natural way just by introducing several tokens in the start place of the product. In the other models discussed, product lot sizes greater than one have to be represented by multiple jobs. In Case B it is shown how to model an operation needing more than one resource. In this way, we can easily handle cases where it is important to take additional resources

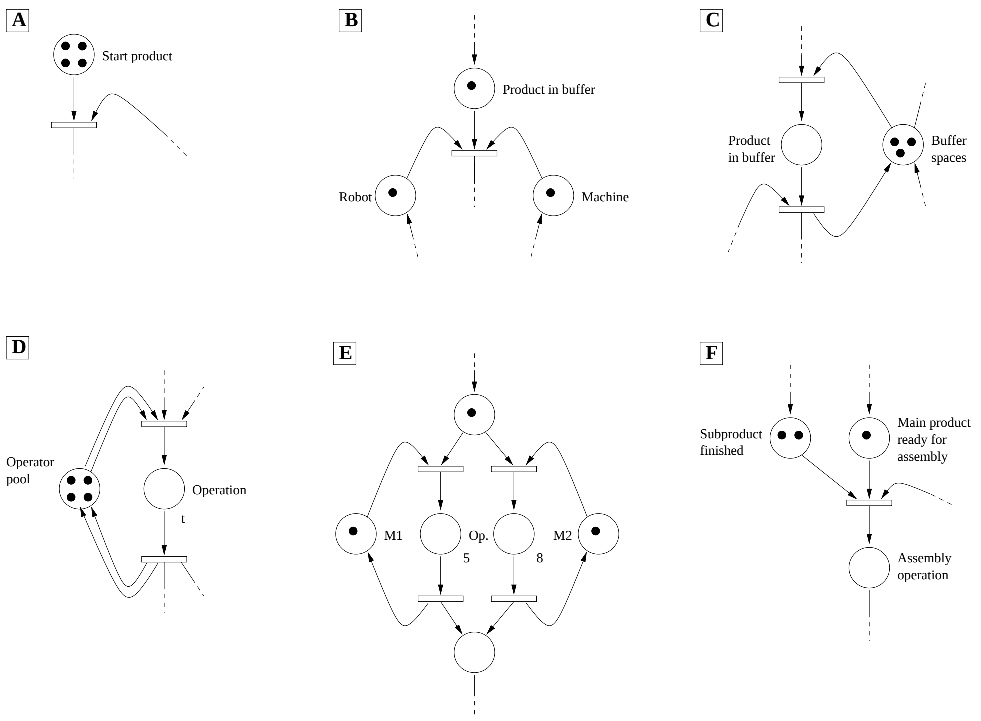

Figure 4.5: Examples of Petri net scheduling model extensions. A: Product lot sizes. B: Multiple resource usage. C: Limited buffers. D: Volumetric resources. E: Routing flexibility. F: Assembly.

such as operators, fixtures and tools into account.

Case C illustrates how to model a finite buffer. A finite buffer is a special case of a *volumetric resource*, i.e. a resource with the capacity of handling more than one operation at a time. A natural extension of this is that some operations demand more than one unit of a volumetric resource. An example of this is shown in Case D. In Case E it is shown how to model the simplest case of *routing flexibility*, i.e. when there are alternative ways of performing an operation. This can easily be generalized to cases with alternative paths having an arbitrary number of operations before the paths join again. Case F shows another application of path joining. It models the situation where sub-products must be available before an assembly operation of another product can be performed.

In Chapter 5 further modelling extensions are made using a non-standard Petri net. We note that for including a modelling feature in a scheduling system, it is not enough that the Petri net allows it. The solution method used must also be able to interpret the modelling feature correctly.

Existing approaches

In this section, an attempt is made to give an extensive historic review of the literature that is relevant for our application in this area. Some of the solution methods discussed below will be presented in more detail in Section 6.2, but we will also give other references.

The earliest examples of production scheduling with Petri nets came in the late 80's, where priority rules were used to make decisions. Shih and Sekiguchi (1991) are the first to use some kind of search in the scheduling process. They present a quite advanced transition-timed Petri net model which takes into account transportation activities (performed by an AGV[6]), limited bufferspace and flexible routing. When a scheduling conflict has to be resolved (i.e. there are two or more conflicting transitions that are enabled or will be enabled in the near future) a decision module is called. The decision module performs a partial *beam search* (see e.g. Winston (1984)) looking ahead some units of time, and then chooses the transition that leads to the partial solution with best evaluation. Simulation experiments done by the authors favour their approach over some simple priority rules when the objective is Just-In-Time. This algorithm is intended as a real-time algorithm that can be used for scheduling and control of a FMS in a dynamic environment, but the partial beam search idea could be interesting as part of some heuristic for static scheduling as well.

Camurri et al. (1993) present a suggestion of how to automatically create a Petri net model from the knowledge base of a FMS. They use colored transition-timed Petri nets where different products can share parts of the net instead of having separate process places like in the example above. Color is then used to distinguish between products in the same place. The authors suggest that a pure Monte Carlo method (they choose the best solution found in a number of trials, where random selection is used to resolve all conflicts in each trial) could be used as a general purpose scheduling strategy for any kind of objective function. For a case study they also develop a special purpose scheduler based on priority rules that can be used in real-time.

In Lee and DiCesare (1994) some of the authors' previous work on FMS scheduling are summarized and extended. They are among the first to employ some kind of intelligent global search to this kind of scheduling problems. They use a place-timed Petri net model which can include features like product lot sizes, multiple resource usage, limited buffers and routing flexibility. With the aim of minimizing makespan, a variant of the *A* search* [7] method (see e.g. Russell and Norvig (1995)) is used as a basis for a set of algorithms. The algorithms differ in the choice of evaluation function for the nodes in the search tree. Apart from one trivial case, none of these functions are *admissible* (see Russell and Norvig (1995)), which is a condition that has to be satisfied if optimality is to be guaranteed during the search. This does not mean that these functions are necessarily bad. Properly designed non-admissible functions reduce the search space explored significantly and can thus be used to produce solutions for problems that are too large for optimal methods. The time needed for producing a solution may still be large however. In order to avoid exploring equivalent branches in the search tree, the previously generated search nodes are checked for a repeated state each iteration. Although this is a costly operation, it pays off globally in most cases. We note that if one wants to guarantee optimality during the search, the comparision between search nodes is not a trivial task. This and many subsequent articles use a simplified comparison without discussing the impact on optimality. We will discuss these issues in more detail in Section 6.1.6.

Sun et al. (1994) present a similar place-timed Petri net model for a FMS, that also includes the control of an AGV system with more than one vehicle. The authors

[6]Automated Guided Vehicle.

[7]The A* search method is perhaps the most important member of the class called *best-first search* methods. Note that many authors use the latter term instead of A*.

present an approach for further reducing search time. They present a *limited expansion A** search method (using one of the evaluation functions of Lee and DiCesare (1994)), that sets a maximum value b for the number of unexplored nodes that are kept in memory. This makes the algorithm similar to a global beam search method, and just as the latter it has for this kind of problem a worst case guarantee for the number of search nodes explored before a solution is found (assuming the system is deadlock-free). This feature makes these approaches more interesting for our application than the pure A* (admissible or not) methods, although it may not be easy to a priori calculate a maximum actual computing time. We note here that Sun et al. (1994) say that the algorithm used in their tests is designed for non-delay scheduling (see Section 3.3.2). This should mean that they have eager-to-fire transitions, however that is not explicitly stated in the article.

In Azzopardi and Lloyd (1994), a depth-first branch-and-bound method for standard job shop scheduling modelled with a place-timed Petri net is presented. Apart from bounding based on the remaining processing times of products, two other search space reduction techniques are incorporated into the algorithm. One is a check for duplicate markings (similar to Lee and DiCesare (1994)), and the other is the removal of alternatives when there are independent transitions enabled at the same time. As far as we can see from the article, the method is restricted to eager firing, i.e. the full scheduling problem is not solved. This is not discussed by the authors.

Jeng et al. (1996) propose a new evaluation function to be used with the A* search method for makespan scheduling. They use an approximate solution to a linear programming problem with the constraints taken from the Petri net state equations, thus incorporating global information into the evaluation function. Although the matrix operations in the new evaluation function make it more costly to compute than the examples from Lee and DiCesare (1994), it is shown with simulation experiments that the method performs better in most cases.

An interesting article is van der Aalst (1996), where it is shown how the problem we call *resource constrained scheduling* (see the end of Section 4.1.3) with arbitrary precedence constraints can be mapped onto a Petri net. No specialized search algorithm is presented, instead is is shown how to analyze (checking for conflicting and/or redundant constraints, calculating lower bounds etc.) and approximately solve a problem using standard Petri net tools. The time model used is more flexible than the standard transition-timed or place-timed counterparts. Time delays are associated with transitions but the completion of the firing is immediate. Instead the individual tokens have a *timestamp* that are set upon firing which equals the earliest time they can be used for inputs to a transition again. This makes it easy to extend the problem with e.g. release dates for products. The price paid for the greater flexibility is a slightly more complicated and memory consuming implementation.

Chetty and Gnanasekaran (1996) model the special case of a *flexible assembly system* using colored Petri nets. In assembly processes it may occur that some operations on a product can be performed in any order, in contrast to the strict technological precedence between operations we usually have in standard manufacturing. The authors show how to model this by representing the status of the product at this stage by several tokens and a so-called *product identity place*. In Section 5.2.5 we show a possible extension of our model to cover such a case, inspired by this article. The scheduling is done with priority rules, but the process is repeated several times with different parameter settings and the best schedule found is returned. The performance of the algorithm is evaluated in simulation experiments.

The idea of using colored Petri nets to obtain a more compact model is explored in Chincholkar and Chetty (1996). The authors represent the production process of a FMS by using only 15 places and 9 transitions. As opposed to the models we have studied, a transition can represent many different events and each place can be "crowded" with tokens being in different states. One example is that a single place represents any product being in the input buffer of any machine. For the scheduling, a priority rule base is used. The performance of the algorithm is evaluated with respect to different objectives by comparing it to simpler priority rules using simulation on a case study. Yan et al. (1997) introduce so-called *extended high level evaluation Petri nets* with a similar modelling idea. They describe a rule-based real-time FMS scheduling system that can repair previously made bad choices by moving products from the input buffer of a machine to the input buffer of another equivalent machine. This kind of models gives a good overview on a high level, but the detailed behaviour is not graphically represented. Also, we believe that the complex data handling associated to the firing of transitions makes the search process considerably slower than an equivalent standard Petri net model. These drawbacks suggest that this modelling approach is not suitable for our particular application.

In Jeng et al. (1997), the authors modify their previous approach (from Jeng et al. (1996)) to work with the objective of minimizing weighted tardiness. In essence, the evaluation function used is based on estimating the minimum remaining processing time for each product and comparing this with the due date. The authors show that their method performs better than some well known priority rules in simulations (but it has of course a much longer execution time). We note however that in the tests made the number of jobs is not very high compared to the number of machines. If this is not the situation, the evaluation function will probably underestimate heavily and we suspect that the method will not work well. In Jeng and Lin (1997), the previously discussed makespan algorithm is improved and applied to the case of manufacturing with assembly operations. The evaluation function used is more advanced than in the authors previous algorithm and simulation experiments favour the new approach.

Based on a previous article from 1996, Xiong and Zhou (1997) presents a hybrid heuristic search algorithm for minimizing makespan in a place-timed Petri net scheduling model that can include multiple resource usage, product lot sizes and limited buffers. The algorithm uses A* search (similar to Lee and DiCesare (1994)) until a certain depth limit is reached in the search tree, then it continues with a *depth-first search* (also called *backtracking search*) until a solution is found. The evaluation function used for the A* search is the maximum operation time left on any of the machines at the system state in question. This function is admissible, which means that the algorithm will provide the optimal solution if it can be found within the specified depth limit. Note however that the function cannot be used without modification in the case of routing flexibility. The algorithm solves the examples presented in Ramaswamy and Joshi (1996) (see Section 4.1.1) with significantly reduced running time.

The same example is examined in Ben Abdallah et al. (1998). The authors present an algorithm with depth-first branch-and-bound as its basis. It is worth noting that from what we can see in the article the search is limited to non-delay schedules (i.e. transitions are eager-to-fire). For the example studied, this reduction of the search space does not cut off the optimal solution, but as stated earlier this is not the general case, a fact not mentioned by the authors. Priority rules are used in order to determine which branch to examine first. Apart from the usual bounding done when examining a search node (based only on path cost) two other tests are made. The first is a check for so-called *empty siphons* (see Murata (1989)) in the net. If such a siphon is found the system will inevitably end up in a global deadlock state, thus the branch can be cut off without further search. The second test checks if the current marking is present in the path of the current minimal solution and if so, the branch is only continued if the time to reach it is better. The example from Ramaswamy and Joshi (1996) is solved even faster than Xiong and Zhou (1997). In this case running with or without siphon checking does not make a difference in the running time, but for the same problem with a large batch size for each product a speed-up of about 15 % is observed with siphon checking.

Proth and Sauer (1998) use a subclass of Petri nets called Controllable-Output nets to model a special case of manufacturing where production volumes are quite high and the flow of different products are constant within the scheduling horizon. The model includes routing flexibility and the solution method suggested uses mathematical programming in a first step in order to distribute the workload on the resources, and then a second step derives a schedule in the net. Although the article describes an interesting combination of Petri net modelling and the use of traditional OR methods, the model is not very useful for our application.

Another approach which connects Petri net techniques with another field of research is described in Richard and Proust (1998). The authors translate a scheduling problem modelled with a subclass of Petri nets into a CLP program (see Section 4.2.1) which is then solved with a standard tool. Examples of minimizing makespan for a number of variants of the flow shop problem is presented. Due to limited modelling power (e.g. routing flexibility is not allowed) and rather long running times, this approach does not seem promising in our case.

In Xiong and Zhou (1998), the authors extend their previous work (see above). The application is the scheduling of a semiconductor test facility, modelled with place-timed Petri nets in the standard manner. The hybrid algorithm from Xiong and Zhou (1997) is compared with one that uses the two methods in the opposite order, i.e. depth-first search (backtracking search) is done up to a certain depth-limit, and then A* search continues the search process from there. These hybrid algorithms are also compared with (pure) A* search and (pure) depth-first search. It turns out that the new hybrid algorithm performs significantly better than the old one, both with respect to objective value and running time. However, if really good results (say within 10 % of optimum) are desired, the examples show that the savings in computing time compared to pure A* search are not significant. The same evaluation function as in Xiong and Zhou (1997) are used in all cases.

Liljenvall (1998) uses results both from Supervisory Control Theory and Operations Research in a place-timed Petri net scheduling model. The work is mainly concerned with standard job shop scheduling problems with no buffers between machines, so avoiding deadlock is one of the main concerns. It is shown how to remove states leading to a (global) deadlock before the search process begins. Several other ways to reduce the search space without sacrificing optimality are presented. The A* search method is used for the scheduling, using evaluation functions based on two-product relaxations and one-machine relaxations. The former is calculated by considering all pairs of products and solving the smaller scheduling problem (again with A*) disregarding all other products for each such pair, and finally using the solution with largest makespan as the estimate. This gives better estimates than by just considering single products, but the computing costs involved are larger and do not scale well to larger problems. The latter relaxation is frequently used in disjunctive graph based methods, and it is computed by solving a preemptive one-machine problem for each machine. This estimate will be better than by just adding up the remaining processing times for the machines (as in Xiong and Zhou (1997)) but requires more time. To handle larger problems, non-admissible evaluation functions are formed by multiplying the admissible estimate with a factor ≥ 1 that decreases with depth in the search tree. Results are presented for some classical job shop scheduling problems, both with and without buffers.

4.3.4 Applicability for our system

Discrete event models are a good candidate for our scheduling system due to the modelling capacity and the ease with which single feasible solutions can be found. For methods that are not rule-based, most research in the area has been made in the context of Petri nets. We have seen examples of many of our desired modelling features in Petri net scheduling approaches, although not all in the same work. The possibility of including all these and the remaining ones in a more advanced model seems to be good.

Chapter 5

A model of the decision support system

We will now present a model for the decision support system described in Chapter 1. As we have already identified the model for individual scheduling scenarios to be the key element of the overall model of the system, we will start with the discussion and the definition of the former. Then we will in a more informal way discuss the global aspects of the application.

5.1 Motivation for the scheduling model selection

We saw in the previous chapter that no previous work known to us fulfills all of the requirements for the scheduling application dealt with in this project. The approaches that we believe have the best chances of meeting our modelling demands are constraint-based methods and discrete event methods. In the former case we expressed some doubts on the ability to guarantee a solution with a short fixed computing time also for large problems, while still having reasonable quality. Also, unless the built-in functionality of a commercial CLP product suffices for most parts of the application, developing a specialized constraint-based scheduler is a huge programming task. A discrete event model is on the other hand fairly easy to develop and to extend. It is also very easy to quickly find a feasible solution. Therefore, with the limited amount of time available for this stage of the project, we decided to concentrate our efforts on a discrete event model.

Another advantage with a discrete event model is that it (as opposed to other approaches) directly describes the actual dynamic behaviour of the system, and can therefore also be used for other purposes than scheduling. Scheduling with discrete event models based on Petri nets has been a growing area of research. Many of our desired modelling features have been included in previous work (although not all in the same), and a variety of solution methods have been been tested. It is of course advantageous to be able to use this research experience, instead of developing an entirely new model. Also, the combination of a graphical representation and a precise mathematical definition that Petri nets have is appealing. As discussed in Chapter 2, there are other graphical modelling techniques that are better than Petri nets with respect to presentation structure and readability. However, there are no standards for building a discrete event model on them. For these reasons we decided to use Petri nets as the starting point for building a model. We note that even if a Petri net model is used for the scheduling, it could be possible to use e.g. an IDEF3 model on top of this for high-level graphical presentations.

5.2 A Petri net model for scheduling

We will now present a Petri net model for scheduling scenarios that meets the minimum requirements stated in Section 1.3.3, as well as a few of the extra modelling features mentioned. For the remaining extra modelling features we will give some suggestions of how to extend the model to include also these. In this section, we will give a slightly informal presentation of the model and illustrate the ideas with figures. In the next section we present a formal notation and a precise definition of the Petri net.

5.2.1 A new timing concept

In Figure 4.5 of the previous chapter, a number of cases are shown that easily can be modelled with a standard place-timed Petri net. In order to include the rest of our required modelling features, we introduce a new timing mechanism in the Petri net. It is an extension of the timing model presented in van der Aalst (1996), where each token in the net has a *timestamp* which equals the earliest time it can be removed from its place by firing a transition. In that model, each transition has a *firing time delay*. When it is fired, new tokens are added instantaneously to the output places, but they will have a timestamp equal to the firing time of the transition plus the delay. We instead associate the time delays with the individual output arcs of the transitions. This allows us to make the Petri net model very compact when we have multiple resources involved in an activity but with different time usage.

There are not only advantages associated with having a more general timing mechanism. In a computer program for a simple place-timed Petri net where we have not more than one token in each place, markings can be stored very efficiently using a bit vector for the token distribution and an integer vector for the timed places. Updates after firing can be made using bitwise operators. This means that the token game executes very fast in such a program and that a tree search method requires a minimum amount of memory. If one allows more than one token except in timed places, the place-timed Petri net can model e.g. product lot sizes larger than one and volumetric resources. Then a bit vector cannot be used any more, instead the number of tokens in each place has to be stored. When we use token timestamps instead to include more modelling features, we in addition have to store a time for each token. This means that execution of the token game is slightly slower[1] and that the memory consumption is larger.

5.2.2 New modelling features

The use of timestamps for tokens makes some features very easy to model:

- Release dates for products. Each token in a product start place can have an individual timestamp (i.e. release date).

[1]By a more or less constant factor. We note that the qualitative behaviour with respect to net size is not affected.

- Repair of a machine. If for example one out of a few parallel machines will not be available directly due to repairs, its associated token can have a timestamp equal to the (expected) time when it can be used again, whereas the other tokens can have a timestamp of 0 at the starting time of the scenario.

- Arrival of a new resource. If a new resource (e.g. an operator) is acquired through some special action of the scheduling scenario, a token is added to the appropriate resource place with a timestamp equal to the arrival time.

- Ongoing activities. An appropriate token distribution and corresponding timestamps can model ongoing activities at the start of the scheduling scenario.

- When tardiness penalties are calculated, one must know the time when individual products are finished. The timestamps of the tokens in the product goal places tell us exactly that.

With a timestamp-based model, a time-consuming activity can in some cases be modelled with a single transition, whereas one has to use an extra timed place and an extra transition in a place-timed Petri net. More complex multi-stage activities with several resources involved can also be compactly modelled when we associate time delays to output arcs. In Figure 5.1, an example of an activity where an operator unloads a product from a machine is shown. First a tool is removed from the machine and becomes available for use at another machine after 3 time units. Then the product is removed so that the machine becomes idle, this takes an additional 2 time units. Finally, after one more time unit, the product is placed in proper storage and the operator can continue with other work. With time delays associated to transitions instead, we would have to use several transitions and places to model such an activity in detail.

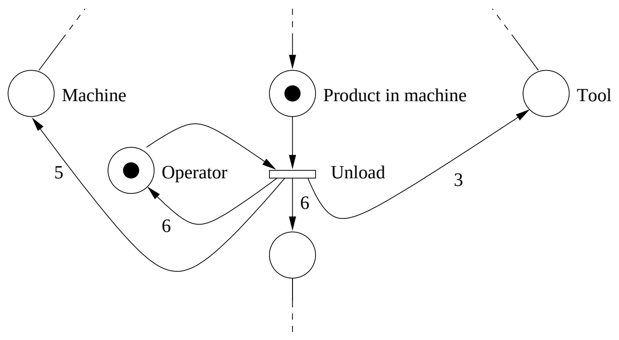

Figure 5.1: Example of an unloading activity.

5.2.3 Modelling the manufacturing process for a product

As in the place-timed Petri net model described in Section 4.3.3, our model has two kinds of places: product (process) and resource places. Each product is represented by an acyclic sub-net of product places and transitions, starting with a product start place and ending with a product goal place. Different product sub-nets are joined by the resource places[2] to form the total Petri net.

There will in general be several activities associated to the processing of a product in a machine:

1. Transport to the machine input buffer. This activity requires the availability of a transport resource and a free buffer space.
2. Loading of the product and setup of the machine. Apart from the machine, an operator must be available and in some cases additional resources such as tools. The machine input buffer space is freed.
3. Processing in the machine. This can either be a fully automatic (in which case the operator is not needed) or a partially manual process.
4. Unloading of the product. This activity requires an operator, and will free the machine and additional resources (see example in Figure 5.1).

If the same resource set is needed for loading, processing and unloading, we can simplify to only one activity represented by a single transition. When the unloading operation is performed, the token representing the product is moved to an intermediate product place (or the product goal place). This intermediate place can be regarded as a buffer place with unlimited capacity. In Section 1.3 the validity of such an assumption for a manned system was discussed. Transports to the next machine input buffer originate from these intermediate places (or product start places), and if there are alternative paths the place in question will be input to more than one transition. Alternative paths forking at an intermediate place (or the product start place) will eventually join in another intermediate place (or the product goal place). When we have such a structure on the Petri net, it will be deadlock free for all reasonable original markings (we do not of course count the goal marking as a deadlock).

We will now illustrate with an example of a product sub-net, see Figure 5.2. Note that loading/processing/unloading is represented with at most two transitions, since the processing is assumed to start immediately after loading. For unloading, one may have to wait for an operator to be available. In the case of two transitions (as in the processing in M1 in the example), the delay of the output arc going to the product place "product in machine" is the sum of the loading/setup time and the processing time. For the second processing step of the example there are two alternatives, M2 and M3. In both these cases, loading/processing/unloading requires both the machine and an operator for the entire process, so they can be represented by a single transition.

5.2.4 The scheduling problem

In order to be able to evaluate a schedule, we introduce two kinds of costs in the Petri net model. The first is costs for performing activities, which we model by allowing a firing cost to be specified for each transition. The second is tardiness penalties. For this we associate a due date and a penalty function with each product goal place. The only restriction we put on the penalty functions are that they are non-decreasing in the tardiness.

[2]Exception: if we have assembly processes, we may have two or more product sub-nets joining into one. Another way of expressing this would be to say that we have one large sub-net for the main product that is to be assembled, and that it has more than one (sub-)product start place.

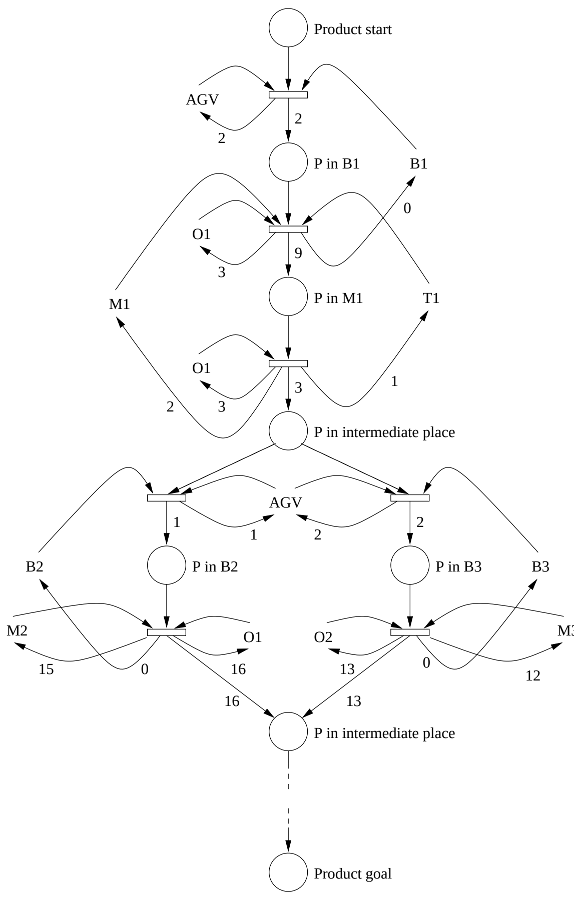

Figure 5.2: Example of a product sub-net. Resource places are shown only with names. Some of the resources may be volumetric resources, i.e. have a maximum number of tokens larger than one. Abbreviations used are P for the product type, M for machine (type), B for a machine input buffer, O for operator(s), T for tool(s), and AGV for a transport resource.

The starting point of the scheduling process is defined by the initial marking (token distribution and timestamps) of the Petri net. Most tokens in the initial marking will be in resource places and product start places, but there can also be tokens in other places representing ongoing activities. A goal marking is one where all product tokens are in the product goal places. A schedule is a sequence of fired transitions which reaches a goal marking from the initial marking. The cost of the schedule is the sum of the firing costs for the transitions[3] in the sequence and the tardiness penalties for each product token in the goal places. The scheduling problem is thus to find the optimal firing sequence with regard to this cost.

[3]We note that this part of the cost is independent of the schedule if we do not have flexible routing.

5.2.5 Possible extensions

If we do not have assembly processes, the number of tokens in product places will be constant during net execution. Another modelling extension that would remove this property is the following: we come to a stage where we have two or more activities to perform, but the order in which we perform them is not specified. Following an idea from Chetty and Gnanasekaran (1996), we can model this case with a Petri net by temporarily splitting the product token into several ones. This is easiest explained graphically, so we refer to Figure 5.3. For simplicity, we let in this example the activities be performed by only a single resource (without preceding transportation to an input buffer), and we omit arc time delays.

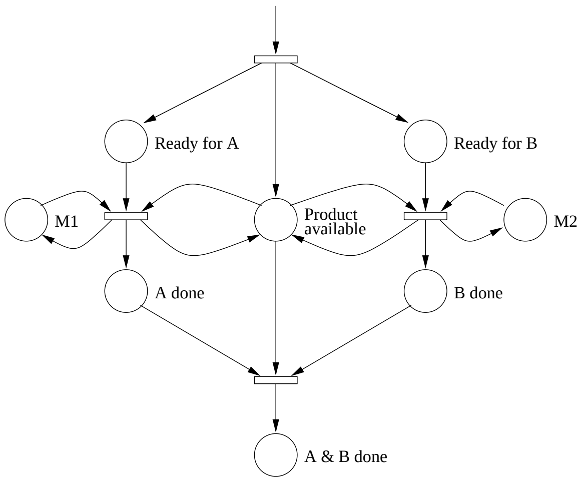

Figure 5.3: Example where two activities A and B can be performed in any order.

Further modelling extensions can be made if we allow tokens to carry more data than just timestamps and transitions/arcs to have special functions (i.e. we take a few steps towards a colored Petri net). One example is sequence-dependent setup times. A machine place token can carry information of its current setting, and time delays for a loading/setup activity may depend on the current setting. When the machine becomes idle after processing, the new setting is stored. Another example is operator skill levels. The tokens in operator resource places may carry information on their skill level. Such a token may only be used as an input to a transition if the skill level allows it. We note that such generalisations of the model will make execution of the token game slower and memory consumption larger in a computer program.

5.3 A formal notation for the Petri net

We will now introduce a formal notation for the properties of the Petri net, that we will use for the rest of this chapter and for describing algorithms. For simplicity, we will restrict ourselves to the case where there is only one input arc per input place of a transition. When needed, we will give a comment of how to extend to the more general case.

We reserve some capital letters for certain meanings throughout the following: P will denote a place (of any type), R a resource place, S a product start place, G a product goal place, J a product internal place, T a transition, I input, O output, E enabling, and M a marking. We will also use n to denote the number of items of some object, c for costs, and t will be used for time. Superscripts are fixed capital letters used to identify the quantity in question. Parameters that can assume different values are shown as arguments inside parentheses. With a marking M we mean not only the positions of all tokens and their timestamps, but also the current time at the marking, denoted by $t(M)$. In other words M includes all data that we need to determine the current state of the system.

5.3.1 Definition of the net structure

For the Petri net we define the following: let the net have the places $P = 1, \ldots, n^P$. Of those, $P^R(i), i = 1, \ldots, n^R$ are the resource places, $P^S(i), i = 1, \ldots, n^S$ are the product start places, $P^G(i), i = 1, \ldots, n^G$ are the product goal places, and $P^J(i), i = 1, \ldots, n^J$ are the product internal places. Also, let the net have the transitions $T = 1, \ldots, n^T$. Transition T has the input places $P^I(T, i), i = 1, \ldots, n^I(T)$, and the output places $P^O(T, i), i = 1, \ldots, n^O(T)$. The corresponding output arc delays are $t^O(T, i), i = 1, \ldots, n^O(T)$. The costs for the operations associated with the firing of transitions are denoted by $c(T), T = 1, \ldots, n^T$.

The other type of costs we include in our model are tardiness penalties. These are always dependent on the problem instance, whereas the operation costs need not be. For a problem instance, we define *due dates* for the different product types. We will associate these with the product goal places and denote them $t^G(G), G = 1, \ldots, n^G$.

5.3.2 Definitions of marking-related properties

For a certain marking M, let place P have $n^M(M, P)$ tokens with timestamps $t^M(M, P, i), i = 1, \ldots, n^M(M, P)$. Without loss of generality, we assume that they are sorted in order of ascending value of the timestamps. The tardiness of a product token with index i in goal place number G we define as

$$D(M, G, i) = \max(t^M(M, P^G(G), i) - t^G(G), 0).$$

The tardiness penalty cost function we denote $c^G(M, G, i)$. It is assumed to have the form $c^G(M, G, i) = c^D(G, D(M, G, i))$ and to be nondecreasing in D for a fixed G. We define the *enabling time* of place P, denoted by $t^{PEM}(M, P)$, in the following way:

$$t^{PEM}(M, P) = \begin{cases} t^M(M, P, 1), & \text{if } n^M(M, P) > 0 \\ \infty, & \text{otherwise} \end{cases}$$

If the enabling time is finite, it will in most cases represent the earliest time a transition that has the place in question as an input place can fire. This is not true in general however, since the firing of another transition might introduce a new token in the place with a lower timestamp.

Note: in the case where there can be more than one input arc going from an input place to a transition, we have to modify the concept of enabling time for a place. We have to instead define enabling time for a place with respect to a transition and compare the number of tokens in the place with the number of input arcs.

We can also define the enabling time of a transition T with respect to a marking M, denoted by $t^{TEM}(M,T)$, according to

$$t^{TEM}(M,T) = \max_{i=1,\ldots,n^I(T)} t^{PEM}(M, P^I(T,i)),$$

i.e. the maximum of the enabling times of the input places of the transition. An enabling time of ∞ thus means that the transition is not enabled at the current marking.

5.3.3 Firing of transitions

When we fire an enabled transition T in a marking M we will reach a new marking, say N. The new current time will be the earliest firing time of T, i.e.

$$t(N) = \max(t^{TEM}(M,T), t(M)).$$

For each input place of T, we remove the first token (i.e. the one with lowest timestamp). If there are multiple input arcs coming from a place, we will of course remove more than one token. New tokens will be added to the output places, having timestamps equal to $t(N)$ plus the output arc delay in question. Then all quantities depending on N are calculated.

5.4 The decision support logic

In this section we will present a suggestion of how the Petri net scheduling model presented above can be used in the decision support system. We will first describe the background data needed, and then we will define types of events that can be handled and the resulting scheduling scenarios examined by the system.

5.4.1 Reserve alternatives

For each product type the information needed to form a product sub-net (see Section 5.2.3) is stored. Apart from the alternatives used in daily production, the system offers the possibility to store *reserve alternatives* for some activities. Examples of such reserve alternatives could be to perform a welding operation manually instead of automatically, or to transport a product using a manual device instead of an AGV. These reserve alternatives are typically slower than the normal ones.

When the decision support system is to be used in a given situation, a Petri net is formed by subnets for all product types that are to be produced within the scheduling time horizon. Reserve alternatives are not added to the subnets unless an unexpected event has made a resource unavailable for whole or a large part of the scheduling time period, and then only for activities where all normal alternatives are affected by the resource in question. The motivation for adding reserve alternatives only when absolutely needed instead of always including them in the model, is that solution methods will generally give the best results when the net is as small as possible.

5.4.2 Extra resources

Ways of getting additional resources to the work center can be stored in the system. This typically corresponds to the temporary hiring of an extra operator, transport device etc. If such an opportunity is available for a certain resource, an associated cost is stored together with the expected delay until the new resource becomes available. In a scenario where the special action of getting an extra resource is considered, an extra token with a timestamp equal to the delay is added to the resource place in question for the initial marking. The extra cost associated is added to the cost of the best schedule found by the solution method.

5.4.3 Repairs

Alternatives for the repair of resources (machines, robots, transport devices etc.) that are malfunctioning can be stored in the system. For each case, the expected time until the resource is repaired is stored together with an associated extra cost. If another resource will be occupied during the repair period, this can be specified as well. This typically corresponds to an operator from the work center doing the repair, in which case the extra cost is zero. For the case of sending for an external repair-man, there will in general be a non-zero extra cost associated but no other resource will be occupied. One must in this case of course also include the expected time until arrival when estimating the time delay until the resource is repaired.

In a scenario corresponding to the special action of performing a certain repair alternative, the appropriate resource place token(s) gets a timestamp equal to the expected time when the repair is done. The extra cost associated to the repair action is added to the best schedule cost. We note that in some environments, repair times may be very uncertain and we suggest that the user should have the possibility to change the relevant expected time delays before the corresponding scenarios are examined.

5.4.4 Acquiring products externally

If it is possible to buy items of a product type from an external supplier, information regarding this can be stored in the system. We need to store a fixed order cost, a cost per item and an expected delivery time. A special action leading to a scheduling scenario will typically be to buy all items of a product type that were to be produced in the scheduling period. In this scenario, the subnet corresponding to this product type will be removed from the total net, and the purchase costs are added to the cost of the best schedule.

5.4.5 Ordering overtime

It should also be possible to store information on time periods where it is possible to have overtime work. If it is possible to add overtime in the scheduling period, it will allow more time before some of the product due dates. In the discrete time scale of the scheduling problem (where off-duty periods are removed) this will have the effect that some due dates are moved forward in time (but in reality they are not of course). For such a scenario, the cost associated to the extra overtime is added to the cost of the schedule.

5.4.6 Events and scenarios

If the decision support system is consulted without any special event occurring, the system will in the simplest case only examine one scheduling scenario where no reserve alternatives are included in the net. In a more advanced version, one might consider some special actions if the outcome of the scheduling is not satisfactory.

When an event has caused the consultation of the decision support system, more than one scenario will in general be examined. As discussed earlier, reserve alternatives may have been added to the product sub-nets as a result of this event. The standard scenario of taking no special action is always considered. For this and other scenarios a check is made before the solution method is started, in order to see if there exists a way of producing every relevant product type. If not, the event has caused a situation that cannot be solved in that scenario and it is unnecessary to start a scheduling algorithm.

We now present some event types together with the scenarios that should be examined in different cases. We suggest that the overtime scenario is examined for all events, unless the event has caused the standard scenario to be unsolvable. We suggest that the system should support the following event types and that the following scenarios should be examined (the standard scenario and the overtime scenario are not mentioned):

- A resource becomes unavailable, possibly for the entire time horizon. This event models among other things a machine breakdown, or an operator that has called in sick. If there are repair alternatives for the resource, one scenario for each such alternative are examined. In these cases the corresponding resource place token will as discussed earlier remain in the net but have a non-zero timestamp. For all other scheduling scenarios, the token is removed entirely from the initial marking. If it is possible to hire an extra resource of the type in question, all such possibilities will result in a scheduling scenario (with an extra resource token added).

- A resource is unavailable until a certain time. This event could be relevant if e.g. a resource is needed somewhere else in the factory for a given time period. In this case, the resource token always remains in the initial marking for all scenarios, but with an appropriate timestamp. The scenarios corresponding to getting an extra resource of the type in question are examined, but not any repair scenarios.

- The release dates of one or more items of a product type are delayed. This event is typically triggered by delayed arrival of some raw material or sub-product needed. If it is possible to buy the product type in question from an external supplier, this scenario is examined.

- The due date of a product type is set tighter. The same scenarios as for the previous event are examined.

- Some new items of a product type are added to the production orders. We suggest again the same scenarios as above.

For the last three event types, it might not in all cases be the product type in question that should be bought from an external supplier, instead it would be better to buy another product type. In Chapter 9, there is a short discussion on such possible improvements in scenario selection.

As discussed in the introduction, the system will allocate a fixed computing time for each scenario. When all scenarios are scheduled, the one with the best total cost (action-related cost + schedule cost) is suggested to the user.

Chapter 6

Algorithms

In this chapter we will present a number of candidate solution methods for our scheduling system. The methods share the property that they are suitable for limited running times. All but the last two presented methods find a first solution very quickly, and the search process can therefore be interrupted at any time without the risk of having no solution at all. The latter two methods must be allowed to finish their search process, otherwise they will produce no complete solution. However, the running time can be approximately controlled by analyzing the problem size and setting some parameters before the search process begins, so these methods are also potentially useful.

6.1 Common elements

Before we describe the individual solution methods in detail, we will look at some aspects of the algorithms that are common to most of them. We will use the notation introduced in the previous chapter.

6.1.1 The limited lookahead heuristic

For a given marking M of the Petri net, we can calculate the enabling time $t^{TEM}(M,T)$ for all transitions $T = 1,\dots,n^T$. We will denote the enabled transitions (i.e. those with $t^{TEM}(M,T) < \infty$) by $T^{EM}(M,i), i = 1,\dots,n^{EM}(M)$, and we assume that they are sorted in order of ascending enabling times. When deciding which transition to fire next, we will have to choose one of these transitions.

We will now introduce a common algorithm time parameter, the so called *lookahead*, denoted by l. Its effect can be loosely stated as follows: when we decide which transition to fire next we will only consider those that have an enabling time that is within the lookahead, counted from the earliest firing time of any of the enabled transitions. The lookahead parameter has a natural interpretation in the physical manufacturing system: it equals the maximum time we allow ourselves to wait until we take the next action when other actions are possible to take.

Formally, we define the role of the lookahead parameter l as follows: for a given marking M and a current time $t(M)$, we only choose between the $n^{EML}(M,l)$ first enabled transitions when deciding which to fire next, where

$$n^{EML}(M,l) = \max i : t^{TEM}(M, T^{EM}(M,i)) \leq \max(t^{TEM}(T^{EM}(M,1)), t(M)) + l.$$

The effect of the lookahead parameter is to reduce the size of the reachability graph. When $l = 0$ we have *eager firing*, and when l is large enough there is no reduction at all of the reachability graph. These two extreme cases are the ones that has received almost all attention in the literature, but with this more general approach we have a whole range of cases. As we will see later, a small lookahead is sufficient for most problems to include optimal paths in the reachability graph, but still gives a significant size reduction. This is beneficial for the efficiency of search algorithms. The suitable setting for l depends on the time scale used and the typical processing times in the manufacturing system.

One might argue that a search algorithm using a good evaluation function (see section 6.1.2) would recognize that firing a transition with a high enabling time (relative to others) is a bad choice anyway, thus removing the need for a limited lookahead. However, we believe there are two strong arguments against this reasoning. The first is that there are many situations where evaluation functions fail to be informative, especially in early stages of the search. The second is that for a search algorithm having only a short computing time available, the reduced number of calls to the evaluation function (which in most cases are computationally heavy) allows it to explore a larger portion of the interesting parts of the search tree during this limited time.

We will in the following sometimes talk of *optimality for a given lookahead*. With this we mean the best possible solution in the reduced reachability graph.

6.1.2 General on objective and evaluation functions

The principles of the solution methods we will present below are independent of the objective of the scheduling problem, which could be minimization of makespan, mean flow time, tardiness penalties, total costs etc. This does not mean that all the algorithms are blind. On the contrary, there is a close connection with the objective function through the so called *evaluation function* that most methods use. Given the current state of the system and the history of actions taken (the transition firing sequence in our case) to reach this state, the evaluation function gives an estimate of the objective value for the best solution reachable from this state. In other words, the evaluation function takes a partial schedule and estimates the value of the best final schedule that can be achieved by completing it. We call the methods that use an evaluation function *informed search methods*, whereas those who do not are referred to as *uninformed search methods*.

The evaluation function typically consists of two parts: an accumulated cost[1] for the actions taken to reach the current state, and an estimation of the remaining cost for reaching the goal state. The latter is calculated using some kind of heuristic method. If this method always gives a lower bound for the remaining cost (i.e. underestimates it) it is called *admissible*. We will call the entire evaluation function admissible if the heuristic estimation is, otherwise we will say that it is *non-admissible*.

Methods like A* and Branch-and-Bound are exact only when admissible evaluation functions are used, otherwise they will be approximate methods. Other tree search methods are approximate by construction (regardless of evaluation function used). The choice between admissible and non-admissible evaluation functions depends among other things on the problem type and the computing time available. If

[1]For simplicity we refer here to the function value type as *cost*, but depending on the objective function used it could be a time related figure as well as actual monetary costs.

there exists a good admissible evaluation function (i.e. one that does not underestimate too much) for the problem, the algorithms (including the approximate ones) will generally benefit from using this rather than a non-admissible one. But for problems where it is difficult to find a good admissible evaluation function, using this may not give much guidance during the search, so a non-admissible function might work better.

6.1.3 Precalculations

In this section we will discuss some calculations that are used for evaluation functions. These calculations use only properties of the net structure, so they can be made before a search process begins. In the presentation, we will exclude two possible modelling features in order to not make things too complicated. The first one is multiple parallel arcs, and the other is assembly (i.e. a joining of process paths without a preceding forking). In many cases, these features do not affect the calculations presented, but in others some modifications have to be done.

Calculation of remaining product processing time

For each process place in the Petri net, we can calculate the minimum remaining time it takes for a token in this place to reach the corresponding goal place. This is simply the sum of all output delays along the shortest path (with respect to these delays) from the place to the goal place. We will denote this quantity $t^P(P)$, and it is easily precalculated for all process places using a standard shortest path algorithm for acyclic graphs (see e.g. the book by Weiss (1995b)) starting a run from each product goal place.

Calculation of remaining resource usage from a process place

Here we will show how to calculate the remaining resource usage of a product, counting from a certain process place P of the product. However, if there are alternative paths, the remaining resource usage depends on the path taken from P to the goal place. In this case it is reasonable to calculate the resource usage along the shortest path (with respect to product processing time) from P, but when using this information we must take into account that this resource usage may not be the actual one. For simplicity, we will present the calculations below for the case of no alternative paths.

For each process place P, we denote the remaining time usage for the resources $t^{PR}(P,R), R = 1,\dots,n^R$. If the resources always are both input and output place of all transitions, this is easily calculated by just summing up output delays for the transitions between P and the goal place. However, it is common that a resource token removed when firing a transition is not returned until another transition further down the product path is fired, and this complicates matters slightly.

For each combination of process place P and resource R we introduce the binary quantity $x(P,R)$ which is 1 if P is "inside" R (i.e. when a token is in P, a token has been removed from R but has not yet been returned) and 0 otherwise. Assume now that we have calculated $t^R(P,R), R = 1,\dots,n^R$ for process place P that is an output place of transition T. This transition has another process place, say Q, as input. For the resources R that are output places of T we set $t^{PR}(Q,R) = t^{PR}(P,R) + t^O(T,i)$ where i is the output place index of R. For those resources R that have $x(P,R) = 1$

we set $t^{PR}(Q,R) = t^{PR}(P,R) + t^{O}(T,j)$ where j is the output place index of P. For all other resources R, we set $t^{PR}(Q,R) = t^{PR}(P,R)$. Using this recursion we can calculate all $t^{PR}(P,R)$ by starting once from each product goal place (where all $t^{PR}(P,R)$ of course are zero). Figure 6.1 should clarify these calculations.

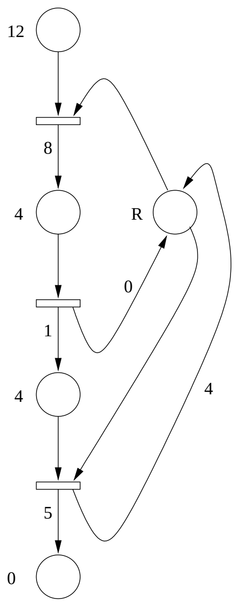

Figure 6.1: Example of calculating remaining resource time for product places. For the product shown, we only show one resource R. The value of $t^{R}(P,R)$ is shown next to the process places.

6.1.4 Calculations depending on the current marking

Here we will use the precalculations described above to calculate some properties of the current marking. These properties are then the basis of different evaluation functions.

Calculation of minimum remaining product processing time

For a given marking M, when we want to calculate the minimum remaining time for a product represented by a token with index i in a process place P, apart from $t^{P}(P)$ we also take into account the waiting time for the product in place P. We will denote this waiting time $t^{W}(M,P,i)$. We define the total remaining time by $t^{PM}(M,P,i) = t^{W}(M,P,i) + t^{P}(P)$.

We will first describe the case when P is input place to only one transition, T (i.e. there are no alternative paths forking from P). We now check this transition's enabling time, $t^{TEM}(M,T)$. If $t^{M}(M,P,i) < t^{TEM}(M,T) < \infty$ and all input resource places R for T with $t^{PEM}(M,R) = t^{TEM}(M,T)$ hold their maximum number of tokens, we set $t^{W}(M,P,i) = \max(t^{TEM}(M,T) - t(M), 0)$. In all other cases, we set $t^{W}(M,P,i) = \max(t^{M}(M,P,i) - t(M), 0)$. What this definition says is that in most cases we will add the waiting time (if any) given by the timestamp of the product token to the processing time of the remaining operations. But if we are certain that the next transition is delayed more because of resource waiting times, we will add this delay instead.

If P is input to more than one transition, we repeat the waiting time calculation described above for each such transition T. Let $i(T)$ be the index of the output place of T that is a process place. Instead of adding $t^P(P)$ to these delays, we add $t^O(T, i(T)) + t^P(P^O(T, i(T)))$. Finally, we will let $t^{PM}(M, P, i)$ be the minimum of these sums.

One advantage with this calculation of minimum remaining product time is that it is relevant for a number of different scheduling objectives, e.g. minimizing makespan, average flow time or tardiness penalties. The main disadvantage is the implicit assumption that the product in question is given top priority and that there is no waiting for the resources needed. If the number of products is large, it is clearly not very realistic to assume that this applies for all of them. For most products, the actual remaining time will be much larger than $t^{PM}(M, P, i)$ in this case.

Calculation of total remaining resource time

For a given marking M, we can estimate the total remaining usage per unit of each resource. We will denote these quantities $t^R(M, R), R = 1, \ldots, n^R$. We will do this iteratively:

1. Set $t^R(M, R) = 0, R = 1, \ldots, n^R$.

2. Go through all process places P. For each token i in P, we first add $t^{PR}(P, R)$ to $t^R(M, R)$ for all resources. If $x(P, R) = 1$, then also add $\max(t^M(M, P, i) - t(M), 0)$ to $t^R(M, R)$.

3. Go through all resource places $P^R(R)$. For each token j in a place, add $\max(t^M(M, P^R(R), j) - t(M), 0)$ to $t^R(M, R)$. Then divide $t^R(M, R)$ by the maximum number of tokens in $P^R(R)$, rounded up (this has of course effect only for volumetric resources).

Unlike the minimum remaining product time calculations, the information gained by making these calculations is relevant also when the number of products are large. As we shall se later, it forms a good basis for an evaluation function that can be used with the objective of minimizing makespan. A disadvantage is that it is not directly applicable to other objectives such as minimizing tardiness penalties, which is crucial to our application. We will see below that it still can be useful as part of other calculations, however.

Note that if we have alternative paths, we do not know if $t^R(M, R)$ will be the actual resource usage for the final part of the schedule (this was also discussed in section 6.1.3), the usage of some resources may be underestimated while it can be overestimated for others. If we were interested in a lower bound for all resources, we would have to make a shortest path calculation for each resource. If the shortest paths would differ much between resources, we would probably end up with an estimate far below values that are possible for the actual schedules. Therefore, we do not believe that this is a good alternative, instead we would accept that the lower bound property does not hold when alternative paths exist.

Calculation of modified remaining product time

When the objective function includes tardiness penalties, we need a good estimate of when individual products are finished. As discussed at the beginning of this section, using the minimum remaining processing time for products does not help us to accomplish this when there are a large number of products, since it does not take the workload of the resources into account. We now present an approach to overcome this problem, based on the idea of modifying the remaining product processing time with a factor calculated from the workload of the resources.

Prior to the search process we can calculate the total remaining resource usage for all places P, denoted by $t^{PR}_{\text{tot}}(P)$. This is done according to

$$t^{PR}_{\text{tot}}(P) = \sum_{R=1}^{n^R} t^{PR}(P, R).$$

For a marking M, we first go through all process places P. For each token i in the place, we calculate $t^{PM}(M, P, i)$. Then we calculate the average of all such minimum remaining product times (we only count those with remaining time > 0), and denote this by $t^{PM}_{\text{avg}}(M)$. We now go through all these tokens again in order to calculate a modified time for the corresponding products. We will call this quantity $t^{PM}_{\text{mod}}(M, P, i, \alpha)$, where $\alpha \geq 0$ is called the *modification parameter*. For a marking M and a token with index i in process place P, we calculate it in the following way:

$$t^{PM}_{\text{mod}}(M, P, i, \alpha) = t^W(M, P, i) + t^P(P) \cdot k_{\text{mod}}(M, P, \alpha), \tag{6.1}$$

where

$$k_{\text{mod}}(M, P, \alpha) = 1 + \alpha \sum_{R=1}^{n^R} \frac{t^{PR}(P, R)}{t^{PR}_{\text{tot}}(P)} \cdot \frac{\max(t^R(M, R) - t^{PM}_{\text{avg}}(M), 0)}{t^{PM}_{\text{avg}}(M)}. \tag{6.2}$$

We will now motivate these formulas. As can be seen, we multiply the remaining processing time with a factor that is greater than or equal to 1. This factor contains a weighted sum of certain terms. The weights are the proportion of the usage of a particular resource to the total resource usage of the product. Note that the weights always sum up to 1. The terms include the difference between the total usage of a resource for all products and the average remaining processing time of the products. If this difference is negative, the resource in question will probably not be a bottleneck, so we do not let this resource affect the factor $k_{\text{mod}}(M, P, \alpha)$. On the other hand, if the difference is positive we will probably have waiting times at this resource, so we let the proportion of the difference to the average remaining product time contribute to the factor.

Finally the weighted sum is multiplied by the modification parameter α, which can be interpreted as (the inverse of) the *priority* of the product. We believe that the most natural choice for α is 1, which corresponds to an estimated worst case for the product (i.e. lowest priority). $\alpha = 1/2$ could be interpreted as medium priority, and a low value as high priority. Note that when $\alpha = 0$, t^{PM}_{mod} reduces to t^{PM}. Of course, values larger than 1 is also possible, which could be used for trying to take into account waiting times that are present for other reasons than those described above.

6.1.5 Examples of evaluation functions

Now we have the building blocks necessary to form evaluation functions for different scheduling objectives.

Minimization of makespan

Although makespan is not the primary objective of our scheduling system, its simplicity still makes it interesting as a test-bench for the algorithms we will describe later. Also, since this is a well studied case, many test problems of different modelling complexity exist.

The first evaluation function, which we will denote $f_1(M)$, uses the minimum remaining product time calculations. We simply take the maximum of all products represented by process place tokens:

$$f_1(M) = t(M) + \max_{P,i} t^{PM}(M, P, i). \tag{6.3}$$

In the second function we present, we use remaining resource times instead:

$$f_2(M) = t(M) + \max_{R} t^{R}(M, R). \tag{6.4}$$

It is now natural to form a third function by combining the two first ones:

$$f_3(M) = \max(f_1(M), f_2(M)). \tag{6.5}$$

All of these evaluation functions are admissible for the scheduling problem with no alternative product paths. $f_2(M)$ and as a consequence also $f_3(M)$ will lose this property if there are alternative paths (the reasons for this were discussed above).

Although intended for use with tardiness penalties, we can of course also use the modified remaining product time calculations here for a fourth evaluation function:

$$f_4(M, \alpha) = t(M) + \max_{P,i} t^{PM}_{\text{mod}}(M, P, i, \alpha). \tag{6.6}$$

Note that $f_4(M, \alpha)$ is non-admissible for all $\alpha > 0$, but not for $\alpha = 0$ since $f_4(M, 0) = f_1(M)$.

Minimization of tardiness penalties

For each product, we will now try to estimate the value of $c^D(G, D)$, and then sum up the costs for all products. For a given marking M and a token with index i in a process place P, let $D_{\text{est}}(M, P, i, \alpha)$ be the estimated tardiness for the corresponding product. It is calculated according to

$$D_{\text{est}}(M, P, i, \alpha) = \max(t(M) + t^{PM}_{\text{mod}}(M, P, i, \alpha) - t^G(G^P(P)), 0),$$

where $G^P(P)$ is the number of the goal place of the process place P. Now we can form an evaluation function by summing over all tokens in process places:

$$f_5(M, \alpha) = \sum_{P,i} c^D(G^P(P), D_{\text{est}}(M, P, i, \alpha)). \tag{6.7}$$

This evaluation function is admissible only for $\alpha = 0$. We believe that values of the modification parameter corresponding to low product priority ($\alpha > 0.5$) will give the best effect, since then products in danger of being delayed can be identified at an early stage.

Minimization of total costs

When we have an algorithm that handles problems with tardiness penalties well, it should not be very difficult to generalize this to the case of total costs. At this stage of our research project, we have focused our efforts on the former case. The latter case will be at the attention of future research within the project. Therefore, here we will only briefly suggest how to make the extension to the total costs case.

In some cases, the extension is trivial. If there are no alternative product paths, the total operation cost will always be the same regardless of schedule, so in effect we are only minimizing against tardiness penalties (we must of course include the operation costs when comparing against other scenarios in a decision support situation). When alternative paths do exist, we have to keep track of the accumulated operation cost during the search process. When estimating the remaining operation cost, our suggestion is to use the costs along the shortest paths of the products with respect to time, since this would agree with the way resource usage calculations are made.

6.1.6 Comparing duplicate markings

We say that two markings are *duplicate* if they have the same number of tokens in every place of the Petri net. More precisely, let M and N be two markings of the net. We call M and N duplicate markings if $n^M(M,P) = n^M(N,P)$ for all $P = 1,\dots,n^P$.

As has been discussed earlier, there are in general many ways to reach a certain distribution of tokens in the net. Some pairs of paths used to reach this point are totally equivalent to each other with regards to the value of the schedule. For other pairs of possible paths, one is clearly better than the other, whereas for other pairs it might be that one path is better in some respects but worse in others. For a solution method that builds a search tree with respect to the transition firing sequence, many of the search nodes in the tree might correspond to a marking that are duplicate with the markings of other nodes. If we analyze this before deciding how to expand the tree next, we may save large amounts of unnecessary work. Therefore, when we are evaluating a newly created search node, we can search for other nodes corresponding to duplicate markings. We then decide which ones of these that are worth continuing from.

We can compare two markings M and N that are duplicate by studying the timestamps of the tokens. First we describe how to make the comparison between markings in the case of makespan objective. We say that M is better than or equal to N if $\max(t^M(M,P,i),t(M)) \leq \max(t^M(N,P,i),t(N))$ for all tokens (P,i). If M is better than or equal to N, there is no point to continue searching from the node corresponding to N, since it will not lead to a better schedule than can be reached from the node corresponding to M. The reverse case of course also applies. If neither M is better than or equal to N or N is better than or equal to M, we cannot say which of the corresponding search nodes that will lead to the best schedule.

When minimization of tardiness penalties is the objective, we compare all tokens that are not in process goal places in the same way as above. For a token i in a process goal place with number G, we instead make the comparison between $\max(t^M(M,P^G(G),i),t^G(G))$ and $\max(t^M(N,P^G(G),i),t^G(G))$. In the total costs case, for M to be considered better than or equal to N, we must also demand that the accumulated cost associated with the firing sequence that lead to M is less than or equal to the one for N.

As mentioned, doing duplicate checking in a search process reduces the number of nodes that have to be expanded at the expense of spending more computing time at each node. Unless the nodes can be stored in an intelligent way, the time spent looking for duplicate markings increases with the size of the search tree.

Liljenvall (1998) has a detailed discussion on comparing duplicate markings for place-timed Petri nets in the case of makespan objective. He points out that it is not enough to look at only the time spent to reach the markings in order to do a proper comparison, one must also look at the remaining times for tokens in timed places. In our model, the comparison described above achieves just the equivalent of that. Some other authors describing algorithms for place-timed Petri nets talk only of comparing the accumulated time, and if their algorithms are implemented accordingly they do not guarantee optimality.

We make an important note here. Although properly performed duplicate checking within a search algorithm does not cut of all optimal paths in the complete reachability graph, this is not always the case in a reachability graph reduced with the limited lookahead heuristic. Consider the following example: assume that we have a token distribution corresponding to a node in the reachability graph through which optimal paths go. Assume also that there are two transitions T_1 and T_2 enabled if we do not take time into account, and that firing T_2 leads to the optimal solution. In a search process with lookahead $l = 2$ we first find a search node corresponding to this token distribution where the enabling times of the transitions are 18 and 20 respectively. Both are thus enabled within the lookahead. Later in the search process, we again arrive at this distribution, but in a faster way so that we have instead enabling times of 17 and 20 respectively. Doing duplicate checking, we will discard the first search node. But in the second search node, T_2 is not enabled within the lookahead and will never be fired from this marking.

For a given lookahead $< \infty$, duplicate checking as we have described it will thus not always preserve optimality for that lookahead, so it must be regarded as a heuristic search space reduction technique in this case. However, duplicate checking will almost always be a good addition to a tree search algorithm for our application. One reason for this is that cases like the example above will probably not occur very often. Another is that even if a duplicate checking algorithm cuts off an optimal solution, it is still very likely that it finds a better solution within our limited computing time than it would without duplicate checking.

6.2 Solution methods

We now have all the necessary ingredients for presenting some solution methods that are candidates for being used in the scheduling system. As mentioned, a method will only have a short computing time available for each scheduling problem it solves. Given such a limited execution time, the method must produce a feasible solution of reasonable quality even if the problem instance is large.

6.2.1 First Enabled

The first method we present is only included for giving a reference solution value that we can compare with the results from other methods. The method constructs only one schedule by beginning from the original marking and then selecting a transition to fire until the goal marking is achieved. When a transition is selected at a marking M, we always take the first one in the list of enabled transitions, i.e. $T^{EM}(M, 1)$.

This means that the First Enabled method gives an eager schedule. We can consider the objective value of the schedule to be an example of the result you can get when the issue of scheduling is not considered in the work center.

6.2.2 Randomized Firing

This method generates as many schedules as it can until the time limit is up and then returns the best one found. Each new trial is started from the original marking, and as in the previous method it then repeatedly selects a transition to fire until the goal marking is reached. When selecting the next transition to fire at a marking M, the limited lookahead heuristic is used to narrow down the number of choices, and then a random selection is made among the remaining $N^{EML}(M, l)$ alternatives.

One major advantage that has to do with the fact that the Randomized Firing algorithm is an uninformed search method is that it can be used with any modelling feature that can be created within our Petri net framework, whereas the informed search methods share the modelling limitations of the evaluation function. Another advantage is that since it uses a minimum amount of calculations before selecting a transition to fire next, the method will have time for quite many trials even for large problems. However, the only "intelligence" in this method is the lookahead heuristic, and besides that the method has to rely on randomization in order to get good results. But although the random selection will make some good choices, in a long transition firing sequence there is a very high probability that there will be a number of bad choices as well. This means that we can expect fairly good but not perfect results from this method.

6.2.3 Best Enabled

This is also a "one-shot method" that we include for reference. Its interest lies in the fact that the algorithms presented below that are based on the branch-and-bound technique will have the Best Enabled schedule as their first solution. This method is similar to First Enabled, but instead of firing the first transition we fire each one that are enabled within the lookahead and calculate the value of the evaluation function of the corresponding new markings. Then we select the marking with the best evaluation (in case of a tie we take the one that lies first in the list) and continue in the same way until we reach the goal marking.

6.2.4 Randomized Best Enabled

A natural extension of Best Enabled to a potentially useful method is to make many trials and to resolve ties between markings having the same evaluation randomly[2]. Since the Randomized Best Enabled algorithm for a decision fires several transitions and makes a call to the evaluation function for each one of them, it will not have time for as many trials as Randomized Firing. Moreover, the number of trials done within the time limit will depend on the lookahead parameter l.

[2]If we have a real-valued evaluation function or an integer valued giving very large values there will probably not be that many ties. In this case it makes sense to adjust the method to select randomly between the choices having evaluation within some interval.

6.2.5 Branch-And-Bound Depth-First-Search

A branch-and-bound method builds a *search tree*, where in our case the nodes in the tree represent markings in the Petri net and the branches correspond to the firing of a transition. The root node in the tree is the original marking. At a certain point in the search process, the method selects a leaf node in the tree (i.e. a node without children) and *expands* this[3]. This is in our case done by firing each enabled transition within the lookahead (with respect to the marking associated to the node) and creating a child node for each marking reached in this way.

Then the child nodes are examined in turn. If the child node corresponds to a goal marking the solution value is calculated, and if this is better than what has previously been found (the best solution value previously found is referred to as the *upper bound*) the transition firing sequence corresponding to the path from the root node is recorded and the upper bound is updated. If it does not correspond to a solution, the evaluation function is called for the associated marking. If the evaluation is higher than or equal to the upper bound, the node is discarded (i.e. the corresponding branch is cut off). If not, the child node is kept for possible future expansion. This process of examining a child node is called *bounding.* If the evaluation function is admissible (i.e. gives a lower bound) the method will never cut off all solutions with optimal objective value, so it is in this case an exact method. We note here that when a candidate node for branching is selected, we again check against the upper bound before creating child nodes, since the upper bound may have changed since the node was created.

We get different variants of the branch-and-bound method depending on how the node selection is performed. We are only interested in node selection strategies that find a first solution quickly. One such strategy is the Depth-First-Search. Here we always select one of the newly created children as the next node to expand, unless we have reached a solution or all children were cut off by the bounding procedure. In the latter cases, we backtrack in the branch until we find an unexpanded node. This means that we always explore all possibilities in a branch before we move on to the next.

The Depth-First-Search is easily implemented using a *stack* data structure. When we create child nodes, we put those who are not cut off on top of the stack. When selecting the next node, we take the one at the top of the stack (removing it from the stack of course). The algorithm is initiated with only the root node on the stack and it is done when the stack becomes empty and no more children are added. We can of course prematurely interrupt the algorithm at any time. Due to the way exploration is made, the stack of unexplored nodes never grows beyond a certain size which depends on the depth of the search tree and the maximum number of branches that can be generated from a node. More details on Depth-First-Search can be found in e.g. Weiss (1995b).

In our case, we put the child nodes on the stack in descending order of their evaluation function value. This means that we will always examine the child node with best evaluation first, and that the first solution we will find will be the one from the Best Enabled method. The Depth-First-Search strategy has both its advantages and drawbacks. A disadvantage is that the method is very sensitive to a bad start, since it takes a long time before it comes back and tries another choice early in the transition firing sequence. On the other hand, the method spends most of its time near the bottom of the search tree where evaluation functions can be expected to

[3]This is also referred to as *branching* from the node.

be most realistic, so it can be expected to do well with the latter part of the firing sequence.

Without duplicate checking

If we do not perform any duplicate checking (see Section 6.1.6), there is no need to keep already explored nodes in memory other than for recursively finding the path from a leaf node to the root. But if we store the entire path in every search node we can delete them right after they have been explored. This together with the bounded stack size makes the Branch-And-Bound Depth-First-Search algorithm without duplicate checking extremely memory efficient, and it can be set to run for a long time without risking memory overflow. However, this advantage is not of much use to us since we are only interested in short execution times. Also, since no duplicate checking is performed we risk exploring many unnecessary paths.

With duplicate checking

If we want to perform a complete duplicate checking during the search, we have to store the already explored nodes in memory. Here we use a simple list to store these search nodes. When we have created a new child node that is not discarded due to bounding, we scan through the other nodes in order to find duplicated markings[4].

First, we go through the list (stack) of unexplored nodes. If we find a duplicate marking, we compare the corresponding node with our new node according to the process described in Section 6.1.6. If it is better than or equal to the new node, we discard the new node and abort the duplicate checking procedure. If the new one is better, we discard the other node and continue to search in the list. If neither node is better than or equal than the other, we keep both and continue the checking procedure.

If the new node was not discarded, we make a similar check with the list of already explored nodes. If the new node survives also this checking procedure, it is added to the list of unexplored nodes.

6.2.6 Branch-And-Bound Best Restart Search

Here we introduce another node selection strategy which we will refer to as *Best Restart Search*, that is also potentially useful for our application. In this search strategy, we follow the Depth-First strategy during the search except for the selection done directly after that one of the following events occurred:

- The previously selected node corresponds to a solution.
- The previously selected node was discarded due to bounding before branching was performed.
- All children of the previously selected node were discarded due to bounding and/or duplicate checking.

[4]Using a *hash table* could speed up this process, especially for large search trees. But with our limited execution time the search tree is not that big, so the savings might not be significant. We have therefore not considered it motivated to investigate this further at the current stage of the project.

In these cases, we instead select the node with the best evaluation from the list of unexplored nodes. This means that we will follow a path towards a goal marking, but when this is reached or a "dead end" is encountered, we restart the search from an entirely different node instead of backtracking on the same path.

This strategy does not share the property of Depth-First-Search that the list of unexplored nodes remains rather small during the search, but for our short execution times this is not a serious problem. Just like in the Depth-First-Search case, we can construct two versions of the method: with and without duplicate checking. For the duplicate checking case a minor modification has to be done. When we discard a node from the list of already explored nodes, we have to keep it in memory in another list, since there is a small possibility that we could find a solution path later in the search that goes through this node. We note that this behaviour was observed a few times in tests when an approximate evaluation function was used.

The Best Restart strategy is not as susceptible to a bad start as Depth-First-Search is, but instead it is not as good at optimizing the latter part of the firing sequence, so it is not easy to predict which one that will do best. The extra time associated with searching for the best node when restarting and the larger memory consumption will in general make it less suitable for optimality proofs than Depth-First-Search, but this does not mean that it will perform worse given a limited execution time on a large-sized problem.

6.2.7 Beam Search

The previously presented informed search methods all have a more or less depth oriented search behaviour, exploring one path at a time. This means that the local decision of choosing the next transition to fire is very important. But the effect of a decision might not show in the evaluation function until a few steps later on, so the guidance on the local level might not be that good. Methods showing more of a *breadth* search behaviour can avoid this drawback by exploring many paths in parallel. However, an exact method with a breadth oriented search strategy will never get to the goal within a limited time for larger problems and are thus useless to our scheduling system.

An heuristic search strategy that at least partially removes the latter mentioned drawback is *beam search.* The method examines each level in the search tree in turn. On a certain level it keeps only a maximum number of search nodes for expansion. This number is referred to as the *beam width* and we will denote it w. All remaining nodes on the current level are expanded, and the best (with regards to the evaluation function used) non-redundant child nodes are kept for the next level, up to a maximum of w nodes. The non-redundancy is achieved by performing duplicate checking between the child nodes and discarding the unnecessary ones.

The Beam Search algorithm cannot be interrupted before it has reached the final level, and this is of course a drawback. However, the running time can fairly easily be controlled a priori by setting an appropriate value for the beam width w. An upper bound for the number of explored nodes during the search is bwd, where b is the maximum branching factor and d is the depth of the search tree. The quantities b and d can be easily calculated for a certain problem instance. The actual running time then depends on how child nodes are sorted with regards to evaluation function value and how duplicate checking is performed. For smaller beam widths, the running time will be nearly linear in w for a fixed problem instance. Empirical testing on the target machine for the scheduling system is probably the easiest way to find appropriate settings.

6.2.8 Limited Expansion A* Search

The A* search method (see e.g. Russell and Norvig (1995) and also Section 4.3.3) is an exact algorithm[5] that also has a breadth oriented search behaviour, but much more intelligent than pure uninformed Breadth-First-Search. The method can be said to be (at least for this kind of problems) a special case of branch-and-bound. The node selection strategy here is to always expand the unexplored node with the best evaluation. The method stops when it has reached a solution, which in the case of an admissible evaluation function will be optimal. With the A* method's search strategy bounding is never needed. Duplicate checking can be performed in the same way as was described for the branch-and-bound methods.

Even though A* can in some respects be proved to be the most intelligent exact search method possible, it still suffers from a worst case execution time that increases exponentially with problem size. By using an appropriate non-admissible evaluation function, solution quality can be traded against a shorter execution time, but the latter is not easy to predict a priori. This fact makes the method unfitted for our application. Sun et al. (1994) present the Limited Expansion A* Search Method for Petri net based scheduling. In this approximate variant of A* search, the number of unexplored nodes kept in memory is limited to a maximum number, which we will denote u. When new unexplored nodes are added, only the ones with the best evaluation function values are kept.

The Limited Expansion A* method has similarly to Beam Search an upper bound *bud* on the number of explored nodes during the search, but the actual running time is not as easy to estimate. One cause for this is that it has not the pure breadth-first behaviour of Beam Search which is very easy to predict. Another is that duplicate checking is performed against already explored nodes, which can make the checking process more and more time-consuming as the search tree grows.

The limited size of the list of unexplored nodes can cause some peculiar behaviour not present in the standard A* search method. As in Branch-And-Bound Best Restart Search, we might find a solution where the path goes through a discarded node, but that potential problem is easily avoided by keeping nodes in memory also after they have been discarded. A more serious situation that can occur is that the list of unexplored nodes gets empty before a solution is reached. This could happen if all new nodes generated gets discarded due to duplicate checking against some already explored nodes, but no unexplored nodes originating from the latter are left in the limited sized list. If the Limited Expansion A* Search Algorithm shows good performance in tests, we suggest that these potential problems are analysed more deeply theoretically and that proper adjustments to the method are made.

[5]If the evaluation function used is admissible.

Chapter 7

Results

We will now evaluate the quantitative behaviour of the algorithms that were presented in the previous chapter. We compare their performance with respect to factors such as problem size, objective function used, choice of evaluation function and settings of parameters by analyzing the results from a range of tests. First, we will present the test problems and discuss the way the tests were performed. Then we will give the test results along with comments and conclusions.

7.1 The test problems and their corresponding Petri net model

7.1.1 An example of a work center

As the basis for the majority of test problem instances, we use an example of a shop floor work center given to us by Prosolvia. It includes a subset of the possible modelling features discussed earlier. The scarce resources of the work center are three machines, the input buffers of those machines, two operators and a transport device (typically an AGV). We will refer to these as M1/M2/M3, B1/B2/B3, O1/O2 and AGV, respectively. Apart from the machine input buffers, the other spaces used for storing products are large enough in order to regard them as infinite. These spaces include the buffers for products waiting to be processed, the buffers for finished products and the spaces used to store products before they are transported to the input buffer of the next machine in their operation list. The latter is usually the output buffer of the last machine that processed the product, but there are also other places of intermediate storage that the operators might use if necessary.

The machines M1 and M2 are operated by O1. The machine M1 is fully automated, so the operator is only needed for the setup of the machine and the removal of the product. For machine M2, the operator is needed during all phases of processing. The same applies to machine M3, which is operated by O2. All transports of products to machine input buffers are handled by the AGV.

The work center is capable of manufacturing four different product types, which we will refer to as P1/P2/P3/P4. The operations required for these are shown in Table 7.1. Transports to machine input buffers are required prior to each of the operations. It is assumed that every transport activity takes 1 time unit. It is also assumed that when an operator performs an unloading activity, the machine is instantly freed, but the operator will spend 1 time unit placing the product in the next storage space, initiating the next transport activity etc. This delay of 1 time unit will of course also apply to the product.

	P1	P2	P3	P4
Op. 1	M1/1/4	M2/2/9	M3/9/1	M1/1/4
Op. 2	M2/2/5	M1/1/11	M1/1/9	M2/2/11
Op. 3	M3/9/11	M3/9/4	M2/2/9	M3/9/8
Op. 4		M1/1/7	M3/9/7	
Op. 5		M2/2/13		
Op. 6		M1/1/5		

Table 7.1: Operation lists for the product types of the work cell. Data are shown in the format “machine/setup time/processing time”.

We can now form a Petri net for the work center according to the model described in Chapter 5. For the operations performed on machines M2 and M3, we can represent them by a single transition, with the output arc delay for operator and product places equal to the sum of setup, processing and unloading times, and with the output arc delay for the machine equal to the sum of only setup and processing. Since O2 and M3 always are used together, we could in principle have replaced them by one single resource. For clarity of the graphical model, we have chosen not to do this. To illustrate the model, we show the part of the Petri net associated with the first product type in Figure 7.1.

7.1.2 Classical standard job shop scheduling problems

We have also tested the algorithms on two well-known standard job shop scheduling problems taken from Fisher and Thompson (1963). These and other test problems are available in OR-Library[1], see Beasley (1990). The first one is a 6 machine, 6 job problem that is fairly easy to solve. It has an optimum makespan of 55. The second is a notoriously difficult 10 machine, 10 job problem that remained unsolved for over 25 years. Nowadays, sophisticated specialized algorithms based on the disjunctive graph model (see Section 4.1.3) can rather quickly reach the optimum, which is 930. However, for discrete event based algorithms designed for much more general problems, it constitutes a tough challenge.

When we make a Petri net description based on our time model for a standard job shop scheduling problem, we do not need the place for a product being in a machine as in the place-timed model that were described in Section 4.3.3. Compared to that case, we can reduce both the number of places and the number of transitions. In Figure 7.2 we show the model for the example from Figure 4.4.

7.2 Notes on the test runs

As was discussed in Section 6.1.5, the algorithm-oriented work of this project has so far been concentrated on makespan and tardiness penalties objectives. Therefore, the test runs presented here have been done on stand-alone scheduling problems. It

[1]Web-site: http://mscmga.ms.ic.ac.uk/info.html

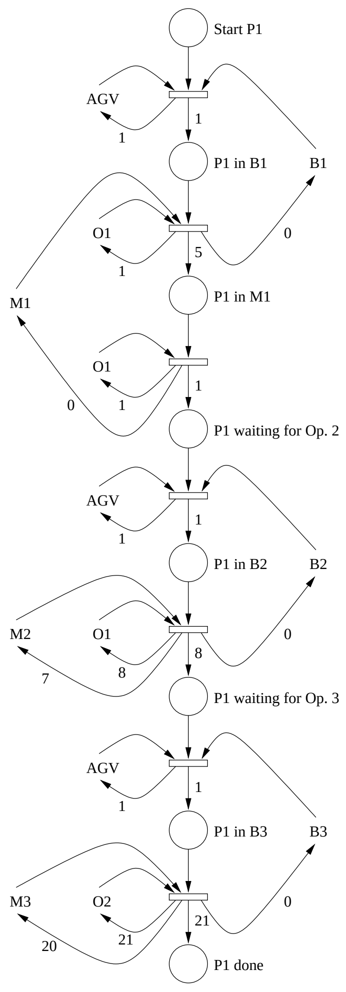

Figure 7.1: The Petri net model for product type 1 of the work center. Resource places are labelled with their names.

is not meaningful to test the algorithms on decision support situations with multiple scheduling scenarios until they are modified to handle the total cost model.

7.2.1 Execution times

When the decision support system is used, it will give a limited amount of computing time to each scheduling scenario. This time will be in the order of a few seconds, say 2 s if there are three or four scenarios. Because of this, we believe that the best way to evaluate the practical usefulness of the solution methods is to use a fixed (and realistic) time limit for all test runs, regardless of problem size. We chose to set this limit to 10 CPU seconds for the test runs. There are two reasons why we did not set the limit as low as e.g. 2 CPU seconds. The first is that a faster computer will probably be used in the real system compared to the one used for the tests (a Sun Ultra 10 workstation, 300 MHz). The second is that since the test versions of

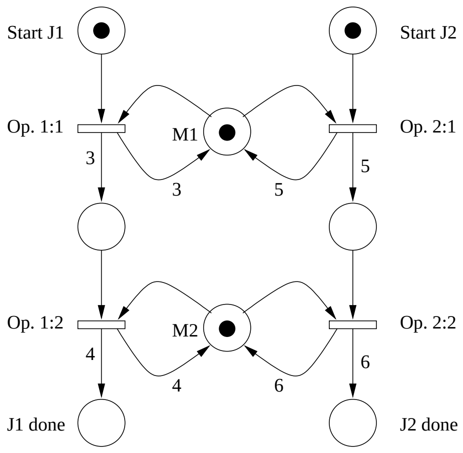

Figure 7.2: The Petri net model for a standard job shop scheduling problem.

the algorithms were developed with ease of implementation as the goal rather than maximum efficiency, they can be much improved with regards to execution time and memory consumption.

7.2.2 Presentation of results

For each test, we present in a table the best solution value found within the 10 CPU seconds time limit by different combinations of algorithms and evaluation functions for different values of the lookahead parameter l. For the randomized algorithms, the average value of 10 runs are presented. If all 10 runs produced the same result, this value is presented without decimals. For the algorithms that do not produce any (complete) solutions until the end of their execution (i.e. Beam Search and Limited Expansion A* Search) and thus cannot be interrupted, different settings of parameters (the beam width w and the max no. of unexplored nodes u respectively) were tested. The solution value corresponding to a setting giving an execution time near the CPU time limit is presented.

In the tables, we use abbreviated names for some of the algorithms:

Randomized for Randomized Firing.

Rand. Best for Randomized Best Enabled.

DFS1 for Branch-And-Bound Depth-First-Search without duplicate checking.

DFS2 for Branch-And-Bound Depth-First-Search with duplicate checking.

Restart1 for Branch-And-Bound Best Restart Search without duplicate checking.

Restart2 for Branch-And-Bound Best Restart Search with duplicate checking.

Beam for Beam Search.

Limited A* for Limited Expansion A* Search.

In connection to the table, we also give some additional test facts regarding the test runs for the different algorithms. These are followed by discussions of the outcome of the test.

7.3 Makespan tests with admissible evaluation functions

In this section, we will present a number of tests based on minimization of makespan using the evaluation functions f_1, f_2 and f_3 that were presented in Section 6.1.5. The test problems do not include alternative product paths, so all of these evaluation functions are admissible.

7.3.1 Test 1. A small number of products.

In this first test, we use a problem based on the work center example in Section 7.1.1. We have only one item of each product type (i.e. a total of 4 products/jobs). We assume that there is one token in each resource place and each process start place, all available from time 0. This means that the in-buffers of the machines have capacity 1. Note that although we have no volumetric resources (e.g. machine in-buffers with capacity larger than one), there are no restrictions against that in any of the algorithms.

Results

Although this scheduling problem is far too big for complete enumeration, it is fairly easy to solve with an intelligent search algorithm. The optimum makespan is 95, which can be found and proved optimal in about 1 CPU second using the DFS2 method with the f_3 evaluation function. The optimum can be found already for a lookahead of $l = 1$. The optimum eager schedule ($l = 0$) is 101. The start evaluation for the f_1 evaluation function is 77, while for f_2 it is 80. As seen, neither of these lower bounds are very sharp, but since the latter bound is better one could expect f_2 to work better than f_1. The test results are shown in Table 7.2.

Additional test facts

- *Randomized.* The algorithm makes around 21400 trials within the time limit, regardless of lookahead. The standard deviation of the 10 runs is below 2 for all lookaheads, and 0 for $l \leq 3$.
- *Rand. Best.* For evaluation function f_1, the algorithm makes between 4670 trials for $l = 0$ down to 3300 trials for $l = \infty$ within the time limit. The corresponding figures for f_2 are 3640 and 2650, and for f_3 they are 3460 and 2470. The standard deviation for the 10 runs is 0 in all cases except for $l = \infty$ and evaluation functions f_2 or f_3.
- *DFS2.* For evaluation function f_1, it takes 145 CPU seconds to prove 95 optimal for $l = \infty$, while for f_2 it takes about 4 CPU seconds.
- *Restart1/2.* For proving optimality, these methods are up to 2 times slower than the DFS counterpart. This does not necessarily mean that it will produce worse solutions within a limited time.

Method	Eval.	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 5$	$l = 10$	$l = \infty$
First Enabled	-	113	-	-	-	-	-	-
Randomized	-	101	95	95	95	95.4	96.8	102.2
Best Enabled	f_1	101	101	101	101	101	101	101
Best Enabled	f_2	113	113	113	113	113	113	113
Best Enabled	f_3	113	113	113	113	113	113	130
Rand. Best	f_1	101	101	101	101	101	101	101
Rand. Best	f_2	101	95	95	95	95	95	96.6
Rand. Best	f_3	101	95	95	95	95	95	95.7
DFS1	f_1	101	96	96	101	101	101	101
DFS1	f_2	113	95	95	95	95	95	95
DFS1	f_3	113	95	95	95	95	95	95
Restart1	f_1	101	101	101	101	101	100	101
Restart1	f_2	101	101	101	101	101	101	99
Restart1	f_3	101	101	101	101	101	101	97
DFS2	f_1	101	95	95	95	95	96	96
DFS2	f_2	101	95	95	95	95	95	95
DFS2	f_3	101	95	95	95	95	95	95
Restart2	f_1	101	95	95	95	95	101	97
Restart2	f_2	101	95	95	95	95	95	95
Restart2	f_3	101	95	95	95	95	95	95
Beam	f_1	101	95	95	95	95	95	95
Beam	f_2	101	95	95	95	95	95	95
Beam	f_3	101	95	95	95	95	95	95
Limited A*	f_1	101	95	95	95	95	95	95
Limited A*	f_2	101	95	95	95	95	95	95
Limited A*	f_3	101	95	95	95	95	95	95

Table 7.2: Results for Test 1.

- *Beam.* For lookaheads 1 and 2, the value of 95 is found already at a beam width of 10 for all evaluation functions, which corresponds only to a couple of hundreds of a second of CPU time. For larger lookaheads, a beam width of 80 is sufficient in most cases (less than 0.5 CPU seconds) to find the best value. The worst case is f_1 with $l = \infty$, where a beam width above 300 (corresponds to about 2 CPU seconds) is needed in order to achieve 95.

- *Limited A**. The algorithm generally needs more time than beam search in order to find the best solution for a given combination of evaluation function and lookahead.

Discussion

The results from this problem does not allow us to draw many conclusions, since it is rather easy to solve. We can however see that resource-based evaluation (f_2) works better than product-based (f_1) in almost all cases for this problem. We also have the indication that a non-zero but small lookahead is good for the results of some algorithms, and that duplicate checking is advantageous for the branch-and-bound algorithms.

7.3.2 Test 2. The Fisher & Thompson 6x6 problem.

This problem, that were presented in Section 7.1.2, are of similar size as the previous one. We can therefore expect most algorithms to do well.

Results

As mentioned earlier, the optimum makespan of this problem is 55. This value can be found and proved optimal in about 3.6 CPU seconds with the Restart2 algorithm using the f_3 evaluation function. A lookahead of $l = 2$ is sufficient to find a value of 55. The optimum makespan for $l \leq 1$ is 57. The start evaluation of evaluation function f_1 is 47, and for f_2 it is 43. This gives a hint that f_1 might give better guidance to the search methods than f_2, as opposed to the case of the previous test. The test results are shown in Table 7.3.

Method	Eval.	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 5$	$l = \infty$
First Enabled	-	68	-	-	-	-	-
Randomized	-	57	57.2	56.4	58.2	63.9	73.5
Best Enabled	f_1	68	68	58	76	76	76
Best Enabled	f_2	68	68	68	68	68	68
Best Enabled	f_3	68	68	59	59	59	59
Rand. Best	f_1	58	58	55	56.7	60.4	70.1
Rand. Best	f_2	57	57	57.3	57.6	58.6	58.7
Rand. Best	f_3	58	58	55	55	55	55
DFS1	f_1	58	60	55	57	57	57
DFS1	f_2	57	60	62	62	62	62
DFS1	f_3	57	58	59	59	59	59
Restart1	f_1	58	58	57	57	71	71
Restart1	f_2	58	60	60	60	60	60
Restart1	f_3	57	58	55	56	56	56
DFS2	f_1	57	58	55	55	55	55
DFS2	f_2	57	57	55	55	58	58
DFS2	f_3	57	57	55	55	55	55
Restart2	f_1	57	58	56	56	56	56
Restart2	f_2	57	57	57	59	61	61
Restart2	f_3	57	57	55	55	55	55
Beam	f_1	57	57	55	55	55	55
Beam	f_2	57	57	55	55	55	55
Beam	f_3	57	57	55	55	55	55
Limited A*	f_1	57	57	59	56	59	69
Limited A*	f_2	57	57	55	55	55	55
Limited A*	f_3	57	57	55	55	55	55

Table 7.3: Results for Test 2.

Additional test facts

- *Randomized.* Within the time limit, the algorithm has time for approximately 28100 trials. The standard deviation varies between 0 for no lookahead up to 1.3 for infinite lookahead.
- *Rand. Best.* For evaluation function f_1, the algorithm makes between 4990 and 2290 trials within the time limit (for $l = 0$ and $l = \infty$ respectively). The corresponding figures for f_2 are 3840 and 1660, and for f_3 they are 3600 and 1510. For f_1, the standard deviation varies between 0 and 1.2, while for f_2 it is below 0.7 for all lookaheads. For f_3 there is no variation at all between runs.
- *DFS2.* For evaluation function f_1, it takes over 10 CPU minutes to prove 55 optimal for $l = \infty$, while for f_2 it takes about 73 CPU seconds, and for f_3 it takes 5 CPU seconds.
- *Restart2.* For proving optimality, this method is sometimes faster than DFS2, but in other cases it is considerably slower.
- *Beam.* For the evaluation functions f_1 and f_3, a beam width of 40 is sufficient to find the best value for almost all values of the lookahead parameter l. This corresponds to a running time of less than 0.4 CPU seconds, even for $l = \infty$. For f_2, a beam width of about 160 is needed to achieve the best result at all lookahead levels. This corresponds to running times up to 2 CPU seconds.

- *Limited A**. For evaluation function f_3, $u = 40$ is sufficient to find the best result for all values of the lookahead parameter except $l = 1$. This corresponds to running times of less than 0.4 CPU seconds. For the other evaluation functions, the algorithm has more trouble finding good solutions, especially for large values of l. In this case, you can also observe that for a fixed lookahead, the result is a bit unstable with regards to the maximum number of unexplored nodes. The solution value does not monotonically improve with increasing value of u, as opposed to what one might expect. One example is that with f_1 and $l = 2$, the method gives 55 for $u = 10$, but 58 for $u = 40$.

Discussion

Like the previous test problem, this problem is not difficult enough to be a real indicator of the differences between the solution methods. For a good choice of evaluation function and lookahead, almost all methods find the optimum value. Beam search continues to show good performance for all evaluation functions and values of the lookahead parameter. As we suspected from the start evaluations, f_1 performs better than f_2 in many cases, although the opposite case is also observed. The combined function f_3 works even better. The indication from the previous test problem that duplicate checking pays off compared to doing no checking, and that high values of the lookahead parameter can sometimes give bad results is supported by these results as well.

7.3.3 Test 3. An average number of products.

We now return to our work center for a problem that should be of average difficulty. We have identical conditions as in Test 1, except that we have more products of each type. To be precise: 4 units of product type P1, 2 units of P2, 2 units of P3 and finally 3 units of P4. This gives a total of 11 products.

Results

The evaluation function f_1 still gives 77 as the start evaluation (remember that it does not account for the workload of the resources), so it cannot be expected to give much guidance for the search functions in the beginning of the decision making process. From f_2, we get 220 as the start evaluation. This turns out to be a very sharp lower bound and this helps the exact methods to find the optimum value in reasonable time despite the fact that the decision tree of this problem is much larger than the one in Test 1. The optimum is 221, which can be found with lookahead 3 and higher. The best eager schedule ($l = 0$) is 236, the best schedule for $l = 1$ is 229 and for $l = 2$ it is 222. The reason for the sharpness of the resource-based lower bound is that the workload of O2 and M3 turns out to be significantly higher than for any other resource. This means that the key for finding a good schedule is to make sure that there are no waiting times at this bottleneck, and since there is enough available time to play with at the other resources this is also possible to do. The results are shown in Table 7.4.

Additional test facts

- *Randomized.* Around 8450 trials are made within the time limit. The standard deviation lies between 0 and 4.5 for the different lookaheads.

Method	Eval.	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 5$	$l = 10$	$l = \infty$
First Enabled	-	240	-	-	-	-	-	-
Randomized	-	236	230.6	229.2	227.9	233.5	244.1	261.2
Best Enabled	f_1	277	277	277	277	289	289	289
Best Enabled	f_2	240	240	240	240	240	240	240
Best Enabled	f_3	240	240	240	240	240	240	240
Rand. Best	f_1	241	238	238	238	238	238.2	238.8
Rand. Best	f_2	236	232.2	229.8	228.6	232.9	237.3	239.7
Rand. Best	f_3	236	231.4	228.5	229.1	231.8	236.7	239.4
DFS1	f_1	242	242	242	242	242	242	250
DFS1	f_2	236	229	229	229	237	237	237
DFS1	f_3	236	229	229	229	237	237	237
Restart1	f_1	241	272	272	273	284	284	284
Restart1	f_2	237	238	230	221	221	221	221
Restart1	f_3	237	238	230	221	221	221	221
DFS2	f_1	242	242	242	242	242	242	242
DFS2	f_2	236	229	222	222	229	226	228
DFS2	f_3	236	229	222	222	229	226	227
Restart2	f_1	237	237	234	223	238	238	239
Restart2	f_2	236	229	222	221	224	224	224
Restart2	f_3	236	229	222	221	224	224	224
Beam	f_1	237	229	245	239	243	229	229
Beam	f_2	236	229	222	221	221	221	225
Beam	f_3	236	229	222	221	221	221	225
Limited A*	f_1	239	240	238	223	230	240	225
Limited A*	f_2	237	231	232	221	221	221	221
Limited A*	f_3	237	231	232	221	221	221	221

Table 7.4: Results for Example 3.

- *Rand. Best.* For evaluation function f_1, the algorithm makes approximately 1570 trials for $l = 0$ and 1080 trials with $l = \infty$. The corresponding numbers for f_2 are 1160 and 740, and for f_3 they are 1080 and 690. For f_1, the standard deviation is below 1 in all cases. For f_2 and f_3 it is higher, up to around 2.5 for some lookaheads.

- *DFS2.* With f_3, optimality of 221 can be proved in about 22 minutes of CPU time.

- *Beam.* Near optimal results are obtained for evaluation functions f_2 and f_3 with $w = 80$ (corresponds to 0.7 - 1.7 seconds of CPU time depending on the value of l), but in order to improve them further several seconds more are needed. Similar as to what was observed with the Limited Expansion A* Method in the previous test, the results can fluctuate when w is increased although the overall trend is towards better results. This effect is largest for lower values (say $w < 100$).

- *Limited A*.* A running time near the time limit is achieved with u near 70 for most cases. The results are slightly unstable with respect to u.

Discussion

We can see that the results for f_2 and f_3 are almost identical, which is natural since f_2 will dominate f_1 for most parts of the search process. The results with f_1 is worse than for f_2 and f_3, but not as bad as one might expect. Beam Search and Limited Expansion A* Search together with the branch-and-bound methods using duplicate checking perform better than the randomized algorithms and the branch-and-bound algorithms without duplicate checking. The behaviour of the algorithms with respect to the lookahead parameter agrees with the results from tests 1 and 2.

7.3.4 Test 4. A large number of products.

We now test a problem with even more products. We have identical conditions as in Test 3, except that we have 6 items of each product type. This gives a total of 24 products. This problem should be large enough to be out of reach for proved optimality with exact methods.

Results

Evaluation function f_2 gives 480 as a lower bound for the problem (f_1 gives 77 as usual), and like in the previous problem this bound turns out to be very sharp. The resources O2 and M3 still make up the bottleneck, although the difference from other resources is not as large as in the previous test. The best solution found is 481 ($l = 2$). For $l = 1$, the best value found is 482 and for $l = 0$ it is 483. We do not know the optimum values for different lookaheads, but judging from the lower bound the best values found must at least be very close to optimum. The results are shown in Table 7.5.

Method	Eval.	$l=0$	$l=1$	$l=2$	$l=3$	$l=5$	$l=10$	$l=\infty$
First Enabled	-	595	-	-	-	-	-	-
Randomized	-	496.2	517.0	521.0	527.8	548.0	587.2	641.7
Best Enabled	f_1	594	594	594	596	628	628	628
Best Enabled	f_2	595	595	595	595	595	595	595
Best Enabled	f_3	595	595	595	595	595	595	595
Rand. Best	f_1	504.0	513.0	514.6	516.0	520.0	526.7	527.8
Rand. Best	f_2	500.7	516.9	518.5	520.0	526.0	536.0	541.3
Rand. Best	f_3	502.2	517.2	520.0	522.8	526.6	534.8	538.7
DFS1	f_1	594	594	594	595	628	628	628
DFS1	f_2	577	580	580	583	591	591	594
DFS1	f_3	577	580	581	583	591	591	594
Restart1	f_1	574	574	574	574	579	559	550
Restart1	f_2	577	577	577	577	577	577	577
Restart1	f_3	577	577	577	577	577	577	577
DFS2	f_1	594	594	594	594	616	616	623
DFS2	f_2	574	575	575	580	585	586	586
DFS2	f_3	574	575	575	580	585	586	586
Restart2	f_1	574	574	574	574	579	559	550
Restart2	f_2	581	581	581	581	581	581	581
Restart2	f_3	581	581	581	581	581	581	581
Beam	f_1	484	482	481	482	490	496	490
Beam	f_2	502	506	511	506	515	515	511
Beam	f_3	502	511	528	515	515	511	519
Limited A*	f_1	504	514	496	525	530	541	533
Limited A*	f_2	522	505	519	508	519	534	506
Limited A*	f_3	520	507	514	510	519	534	519

Table 7.5: Results for Test 4.

Additional test facts

- *Randomized.* Around 3680 trials are made within the time limit. The standard deviation lies between 2 and 4 for all values of l except ∞ where it is as high as 10.

- *Rand. Best.* With evaluation function f_1, the algorithm has time for approximately 610 trials for $l = 0$ and 410 trials with $l = \infty$. With f_2 the corresponding numbers of trials are 400 and 240, and for f_3 they are 370 and 230. The standard deviation is typically between 3 and 5.

- *Beam.* The beam width that gives a running time near the limit varies from $280 - 300$ for $l = 0$ down to $140 - 170$ for $l = \infty$ (the low values are for f_3 and the higher are for f_1).
- *Limited A*.* Values between 20 and 30 for u give a running time near the time limit.

Discussion

The results from this problem are very interesting. They show that the branch-and-bound algorithms cannot compete with the other algorithms in this case, most likely due to the large size of the problem. The most surprising results are perhaps the success of Beam Search with evaluation function f_1, in spite of the fact that the search process is more or less blind at the early stages. We do not believe however that this is representative of the behaviour of f_1 with a large number of products, it is most likely a "strike of luck". Another result that might be better than expected is the one from Randomized Firing with $l = 0$, beaten only by results obtained with Beam Search. We also note that as in the previous test, the results using f_2 and f_3 are very similar.

7.3.5 Test 5. The Fisher & Thompson 10x10 problem.

As mentioned earlier, this is a problem where it is hard to find good solutions and lower bounds, even for specialized algorithms. Therefore, with our limited computing time we cannot expect near-optimal results from discrete event based algorithms designed for much more general problems. However, it is interesting as a difficult test case for comparing our different algorithms relative to each other.

Results

The start evaluation from f_1 is 655, and from f_2 it is 631. These lower bounds are very far from the optimal value 930. The results are shown in Table 7.6.

Additional test facts

- *Randomized.* Around 3050 trials are made within the time limit. The standard deviation lies in the range from 8 to 23, except for $l = \infty$ where it is around 40.
- *Rand. Best.* With evaluation function f_1, the algorithm has time for approximately 520 trials for $l = 0$ and 180 trials with $l = \infty$. With f_2 the corresponding numbers of trials are 460 and 130, and for f_3 they are 430 and 120. The standard deviation is typically between 5 and 20 for small lookaheads, while it can be larger for higher values of l.
- *Beam.* The beam width that gives a running time near the limit is around 280 for $l = 0$ and 100 for $l = \infty$. The results can be a bit unstable with regards to beam width for a given lookahead. An example is that with f_2 and $l = 0$ or $l = 2$, $w = 40$ gives 993 whereas beam widths around 300 give 1020-1030.
- *Limited A*.* Values between 25 (high lookaheads) and 65 (small lookaheads) for u give a running time near the time limit. The results vary quite a lot with increasing u for a given value of l, especially for large values of the latter.

Method	Eval.	$l = 0$	$l = 2$	$l = 5$	$l = 10$	$l = 20$	$l = 50$	$l = \infty$
First Enabled	-	1262	-	-	-	-	-	-
Randomized	-	1030.3	1033.8	1041.2	1058.6	1118.2	1385.1	1892.1
Best Enabled	f_1	1092	1095	1099	1118	1121	1526	1526
Best Enabled	f_2	1262	1262	1262	1262	1262	1262	1262
Best Enabled	f_3	1126	1126	1126	1132	1124	1411	1411
Rand. Best	f_1	1046	1047.8	1040.6	1055.9	1094.2	1342.0	1475.8
Rand. Best	f_2	1062.2	1067.7	1062.0	1074.8	1090.3	1134.1	1146.6
Rand. Best	f_3	1044.8	1045.8	1045.9	1056.6	1073.0	1101.5	1114.7
DFS1	f_1	1092	1094	1094	1118	1097	1526	1526
DFS1	f_2	1190	1190	1190	1190	1190	1262	1262
DFS1	f_3	1126	1126	1126	1126	1107	1411	1411
Restart1	f_1	1046	1059	1099	1071	1095	1410	1526
Restart1	f_2	1205	1205	1205	1205	1205	1205	1205
Restart1	f_3	1100	1105	1100	1132	1090	1386	1411
DFS2	f_1	1087	1092	1094	1118	1097	1450	1486
DFS2	f_2	1130	1130	1130	1130	1190	1214	1254
DFS2	f_3	1072	1081	1126	1107	1107	1411	1411
Restart2	f_1	1045	1047	1058	1073	1107	1435	1526
Restart2	f_2	1205	1214	1245	1245	1245	1245	1245
Restart2	f_3	1037	1081	1060	1056	1075	1150	1150
Beam	f_1	1023	1030	1023	1050	1011	1149	1169
Beam	f_2	1024	1029	1028	1035	1057	1042	1110
Beam	f_3	991	991	991	979	1053	1123	1065
Limited A*	f_1	1020	1065	1065	1045	1153	1482	1453
Limited A*	f_2	1042	1088	1046	1155	1142	1158	1211
Limited A*	f_3	985	1058	1064	1052	1106	1516	1556

Table 7.6: Results for Test 5.

Discussion

For this problem, f_1 works better than f_2 in general, and as in previous tests the combined function f_3 has a good behaviour in most cases. As in the previous large-sized problem, the branch-and-bound methods do not work well. We can see that Beam Search performs much better than the other methods. The randomized algorithms show fairly decent and stable results for small lookaheads, whereas the Limited Expansion A* Search method gives a few good results but also some bad ones.

Other authors have also examined this problem using Petri Net methods. van der Aalst (1996) finds a schedule with makespan 1023 by heuristically adding extra precedence constraints to the problem and then generating solutions with a standard Petri net tool. The running time was in the same range as in our case. Liljenvall (1998) uses an A* search method with advanced evaluation functions and search space reduction techniques, and obtains as best a result of 959 (with a few minutes of running time) which is better than our best result (979). However, many of the features used are difficult to generalize to a more general scheduling model and/or computationally intensive, so they are not so interesting for our particular application. This paper was also discussed in Sections 4.3.3 and 6.1.6.

7.3.6 Conclusions

An observation we can make is that the makespan of the First Enabled method lies 9 - 29 % above the best result obtained with other methods for the different problems. This indicates how much you can gain for this objective by using scheduling methods. From the results of these makespan problems, we can draw some conclusions that can be of interest also for other objectives. The issue of whether resource based or product based evaluation is best has no clear answer, since we saw that this varied from problem to problem. An evaluation function combining both should take away this problem-dependence and eliminate a potentially bad behaviour, and that is exactly what has been observed with the function f_3. In many cases, it even works better than the best of f_1 and f_2 for that problem. The conclusion we can draw from this is that we should focus both on products and resources when designing an evaluation function.

We can also tell from the results that a limited lookahead is advantageous for all solution methods. Among the algorithms, Beam Search is the one that gives the overall best results. It works well even when the evaluation function used is not very good for the problem. The branch-and-bound methods with duplicate checking combined with the best evaluation function give good results for the small and medium sized problems, but for larger problems they perform badly. There are no clear indications as of whether the depth first or the best restart strategy is best given a limited computing time, but the former seems to be the most efficient when large parts of the search tree are to be explored. The Randomized Firing algorithm gives fairly good results for all problems when a small lookahead is used, but as expected it does not give good results for larger lookaheads. The Randomized Best Enabled method performs slightly worse than the former for the larger problems, but it is not as sensitive to the setting of the lookahead parameter l. The Limited Expansion A* Search method shows a bit of an unstable behaviour: really good results in some cases but bad ones in others.

7.4 Makespan tests with a non-admissible evaluation function

In this section, we will present results from the same test problems as in the previous section, but using the non-admissible evaluation function f_4 (see Section 6.1.5). Although the modified product time calculations are intended for the case of tardiness penalties or total costs, it is interesting to verify that it does not perform badly in the case of makespan objective. For reference we present again the results with evaluation function f_1 since this corresponds to the case when the modification parameter α is zero.

7.4.1 Test 6. A small number of products.

Here we present the results for the problem of Test 1.

Results

The start evaluation with $\alpha = 1$ (which corresponds to the worst case estimation) is 97 (recall that the optimum is 95), which is a slight overestimate but much closer to the optimum than the lower bounds given by the admissible evaluation functions. The test results are shown in Table 7.7.

Additional test facts

- *Rand. Best.* The algorithm makes between 2810 trials for $l = 0$ down to 2140 trials for $l = \infty$ within the time limit for $\alpha > 0$. As can be seen from the table, there are no variations in the results.

Method	α	$l=0$	$l=1$	$l=2$	$l=3$	$l=5$	$l=10$	$l=\infty$
Best Enabled	0	101	101	101	101	101	101	101
Best Enabled	0.25	101	106	106	106	101	101	101
Best Enabled	0.5	101	107	107	107	101	101	118
Best Enabled	0.75	101	107	107	107	101	101	118
Best Enabled	1	101	107	107	107	107	107	118
Rand. Best	0	101	101	101	101	101	101	101
Rand. Best	0.25	101	101	101	101	101	101	101
Rand. Best	0.5	101	101	101	101	101	101	101
Rand. Best	0.75	101	101	101	101	101	101	101
Rand. Best	1	101	101	101	101	101	101	101
DFS1	0	101	96	96	101	101	101	101
DFS1	0.25	101	96	96	101	101	101	101
DFS1	0.5	101	107	103	103	101	101	102
DFS1	0.75	101	107	107	107	101	101	102
DFS1	1	101	107	107	107	107	107	106
Restart1	0	101	101	101	101	101	100	101
Restart1	0.25	101	101	101	101	101	101	101
Restart1	0.5	101	107	107	107	101	101	103
Restart1	0.75	101	107	107	107	101	101	101
Restart1	1	101	107	107	107	107	107	102
DFS2	0	101	95	95	95	95	96	96
DFS2	0.25	101	96	96	96	96	96	96
DFS2	0.5	101	102	102	103	96	101	102
DFS2	0.75	101	107	107	107	101	101	102
DFS2	1	101	107	107	107	107	107	102
Restart2	0	101	95	95	95	95	101	97
Restart2	0.25	101	95	95	95	96	101	97
Restart2	0.5	101	100	100	100	96	96	101
Restart2	0.75	101	107	107	107	101	101	101
Restart2	1	101	107	107	107	107	107	102
Beam	0	101	95	95	95	95	95	95
Beam	0.25	101	95	95	95	95	95	95
Beam	0.5	101	95	95	95	95	95	95
Beam	0.75	101	95	95	95	95	95	95
Beam	1	101	95	95	95	95	95	95
Limited A*	0	101	95	95	95	95	95	95
Limited A*	0.25	101	95	95	95	96	96	97
Limited A*	0.5	101	95	95	95	96	96	98
Limited A*	0.75	101	95	95	95	96	100	97
Limited A*	1	101	95	95	103	96	100	102

Table 7.7: Results for Test 6.

- *DFS2, Restart2.* For $\alpha > 0$, the solutions are given very quickly for all but the highest lookaheads. This is related to the large amount of cutting made in the search tree.
- *Beam.* A beam width of 80 is sufficient to find the best results for all lookaheads except the largest ones. This corresponds to running times of less than 0.5 CPU seconds.
- *Limited A*.* In many cases, the list of unexplored nodes never gets full over a certain size of u, so improving u further does not change the result or affect the running time.

Discussion

For many of the methods, the results get worse as α increases. For the branch-and-bound methods, this is probably due to the fact that branches in the search tree get cut off due to overestimation even though they contain a potentially good solution. Also, the negative trend seen in Best Enabled suggest that the local decision of choosing the next transition to fire is not helped by the modified estimation. However, the most important result is that the evaluation function works well with the best algorithm from the previous series, namely Beam Search. Limited Expansion A* Search also does well for lower values of l.

7.4.2 Test 7. The Fisher & Thompson 6x6 problem.

This is the same problem as in Test 2.

Results

The start evaluation with $\alpha = 1$ is 55, which is exactly the same as the optimum makespan. The test results are shown in Table 7.8.

Method	α	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 5$	$l = \infty$
Best Enabled	0	68	68	58	76	76	76
Best Enabled	0.25	68	68	61	61	79	79
Best Enabled	0.5	68	68	58	76	78	78
Best Enabled	0.75	65	66	59	63	63	65
Best Enabled	1	70	76	76	76	76	79
Rand. Best	0	58	58	55	56.7	60.4	70.1
Rand. Best	0.25	58	58	55	58.2	59	59
Rand. Best	0.5	58	58	57	58.7	61	61
Rand. Best	0.75	58	58	57	59.6	61	61
Rand. Best	1	62	71	71	71	72	74
DFS1	0	58	60	55	57	57	57
DFS1	0.25	58	61	55	57	56	56
DFS1	0.5	59	62	58	57	57	57
DFS1	0.75	58	60	59	58	58	58
DFS1	1	61	76	67	70	70	70
Restart1	0	58	58	57	57	71	71
Restart1	0.25	58	58	58	59	65	64
Restart1	0.5	59	61	58	61	64	64
Restart1	0.75	61	61	59	61	63	59
Restart1	1	65	66	58	59	62	62
DFS2	0	57	58	55	55	55	55
DFS2	0.25	57	59	55	55	56	56
DFS2	0.5	59	59	58	55	55	55
DFS2	0.75	58	60	59	58	58	57
DFS2	1	61	64	58	66	67	70
Restart2	0	57	58	56	56	56	56
Restart2	0.25	59	58	58	56	58	58
Restart2	0.5	59	59	58	56	58	58
Restart2	0.75	59	61	59	57	59	59
Restart2	1	65	59	58	59	57	59
Beam	0	57	57	55	55	55	55
Beam	0.25	57	57	55	55	55	55
Beam	0.5	57	57	55	55	55	55
Beam	0.75	57	57	55	55	55	55
Beam	1	57	57	55	55	55	55
Limited A*	0	57	57	59	56	59	69
Limited A*	0.25	57	57	55	55	55	58
Limited A*	0.5	59	59	57	55	55	55
Limited A*	0.75	59	59	58	55	55	55
Limited A*	1	61	59	58	58	55	55

Table 7.8: Results for Test 7.

Additional test facts

- *Rand. Best.* For $\alpha > 0$, the Randomized Best Enabled algorithm has time for between 2780 and 1200 trials within the time limit, depending on lookahead.

- *Beam.* A beam width of 80 is sufficient to find the best results for almost all lookaheads. This corresponds to running times of up to 1 CPU second.
- *Limited A*.* As in the previous test, the behaviour of the method is unchanged above a certain value of u in many cases.

Discussion

The conclusions from the last test is supported also by this one. Beam Search and to some extent Limited Expansion A* Search works well with evaluation function f_4, whereas the other algorithms tend to get worse results with increasing value of α.

7.4.3 Test 8. An average number of products.

This is the same problem as in Test 3.

Results

The start evaluation for $\alpha = 1$ is 242, which is a slight overestimate of the optimal schedule which has a makespan of 221. But it is of course much more realistic compared to 77 which is the estimation without modification ($\alpha = 0$). The results are shown in Table 7.9.

Additional test facts

- *Rand. Best.* For $\alpha > 0$, the Randomized Best Enabled algorithm has time for between 780 and 560 trials within the time limit, depending on lookahead.
- *Beam.* Already with $w = 20$ (less than 0.5 CPU seconds), results near the final ones are achieved. One has to go to quite high beam widths, giving a running time near the limit, in order to improve the values much from this.
- *Limited A*.* Values for u in the range 35 to 60 give running times near the limit in most cases.

Discussion

In this test, the value of α does not have that much impact on the results. Beam Search again performs best, although the advantage over the other methods is not as evident as in the other cases. It is interesting to note that almost all results are quite close to the start evaluation of 242 obtained with $\alpha = 1$.

7.4.4 Test 9. A large number of products.

We now test the modified evaluation function on the problem of Test 4.

Results

With $\alpha = 1$, the start evaluation is 527 which is to be compared with the best known schedule which has a makespan of 481. This estimation is again fairly realistic. In Table 7.10, the test results are shown.

Method	α	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 5$	$l = 10$	$l = \infty$
Best Enabled	0	277	277	277	277	289	289	289
Best Enabled	0.25	277	277	277	277	295	295	295
Best Enabled	0.5	276	276	276	276	299	294	299
Best Enabled	0.75	288	276	276	276	299	258	258
Best Enabled	1	288	294	294	294	294	248	276
Rand. Best	0	241	238	238	238	238	238.2	238.8
Rand. Best	0.25	241	241	241	241	241	241	241
Rand. Best	0.5	241	241	241	241	241	241.1	241.4
Rand. Best	0.75	241	241	241	241	241	241.2	242.0
Rand. Best	1	241	241	241	241	241.1	242.5	248.0
DFS1	0	242	242	242	242	242	242	250
DFS1	0.25	242	242	242	243	250	250	268
DFS1	0.5	241	241	241	242	246	249	274
DFS1	0.75	241	241	241	243	246	243	244
DFS1	1	241	242	241	241	241	248	261
Restart1	0	241	272	272	273	284	284	284
Restart1	0.25	241	273	272	272	291	291	291
Restart1	0.5	241	241	241	241	239	245	245
Restart1	0.75	242	241	241	241	246	249	247
Restart1	1	241	259	241	241	241	248	251
DFS2	0	242	242	242	242	242	242	242
DFS2	0.25	242	242	242	243	244	244	244
DFS2	0.5	241	241	241	242	246	241	246
DFS2	0.75	241	241	241	241	246	243	244
DFS2	1	241	253	241	241	241	245	249
Restart2	0	237	237	234	223	238	238	239
Restart2	0.25	237	242	235	242	266	245	248
Restart2	0.5	241	241	241	241	239	244	254
Restart2	0.75	242	241	241	241	246	247	258
Restart2	1	241	258	241	241	241	248	251
Beam	0	237	229	245	239	243	229	229
Beam	0.25	237	237	237	237	241	234	235
Beam	0.5	245	237	237	237	241	234	234
Beam	0.75	237	237	237	237	241	234	231
Beam	1	237	236	237	237	241	234	234
Limited A*	0	239	240	238	223	230	240	225
Limited A*	0.25	237	239	232	223	249	261	264
Limited A*	0.5	246	237	237	238	256	258	259
Limited A*	0.75	237	241	243	247	256	247	277
Limited A*	1	241	241	244	247	247	244	244

Table 7.9: Results for Example 8.

Additional test facts

- *Rand. Best.* For $\alpha > 0$, this method can make between 260 and 190 trials within the time limit, depending on lookahead. The standard deviation lies between 2 and 11.
- *Beam.* Beam widths from 110 ($l = \infty$) up to 220 ($l = 0$) give a running time near the limit.
- *Limited A*.* The values for u that give execution time near the limit vary between 14 and 34.

Discussion

The most striking thing about these results is that the Limited Expansion A* Search method fails to find any schedule in some cases. The possibility of such occurrences was described in Section 6.2.8. This suggests that this algorithm has to be modified to cope with such cases if it is to be used in a scheduling system. The effect of the product time modification varies between the algorithms. The branch-and-bound algorithms generally give better results with $\alpha > 0$ than without modification, whereas the

Method	α	$l=0$	$l=1$	$l=2$	$l=3$	$l=5$	$l=10$	$l=\infty$
Best Enabled	0	594	594	594	596	628	628	628
Best Enabled	0.25	563	571	576	588	611	611	608
Best Enabled	0.5	553	545	544	556	562	563	570
Best Enabled	0.75	529	547	543	548	577	630	630
Best Enabled	1	569	581	575	572	586	685	703
Rand. Best	0	504.0	513.0	514.6	516.0	520.0	526.7	527.8
Rand. Best	0.25	514.4	527.8	532.8	535.8	536.9	545.2	548.4
Rand. Best	0.5	514.3	533.6	533.8	540.0	547.7	559.1	561.1
Rand. Best	0.75	541.6	550.4	547.6	552.4	561.4	587.3	588.7
Rand. Best	1	551.6	558.4	560.2	563.4	571.1	616.2	620.9
DFS1	0	594	594	594	595	628	628	628
DFS1	0.25	563	571	576	588	606	606	608
DFS1	0.5	553	521	521	531	547	563	570
DFS1	0.75	518	543	539	537	556	630	630
DFS1	1	547	546	565	566	564	685	703
Restart1	0	574	574	574	574	579	559	550
Restart1	0.25	549	567	576	577	594	578	604
Restart1	0.5	513	519	521	531	547	563	570
Restart1	0.75	513	543	539	528	573	602	605
Restart1	1	538	563	563	561	561	628	660
DFS2	0	594	594	594	594	616	616	623
DFS2	0.25	528	551	568	585	585	585	585
DFS2	0.5	523	518	518	520	525	526	532
DFS2	0.75	518	529	534	533	550	609	606
DFS2	1	548	553	552	556	564	671	690
Restart2	0	574	574	574	574	579	559	550
Restart2	0.25	549	567	566	577	594	578	577
Restart2	0.5	513	519	521	531	547	551	570
Restart2	0.75	513	543	539	548	573	605	605
Restart2	1	538	563	563	561	561	628	660
Beam	0	484	482	481	482	490	496	490
Beam	0.25	491	514	505	502	532	495	530
Beam	0.5	507	522	522	530	513	532	540
Beam	0.75	503	510	539	538	553	552	550
Beam	1	518	525	536	545	544	572	560
Limited A*	0	504	514	496	525	530	541	533
Limited A*	0.25	517	505	521	511	∞	517	511
Limited A*	0.5	494	521	513	517	541	∞	574
Limited A*	0.75	541	∞	∞	∞	538	561	581
Limited A*	1	507	∞	∞	∞	587	575	584

Table 7.10: Results for Test 9.

results from Randomized Best Enabled, Beam Search and Limited Expansion A* Search tend to get worse when α is increased. Recall however that Beam Search with evaluation function f_1 ($\alpha = 0$) performed suprisingly well for this problem.

7.4.5 Test 10. The Fisher & Thompson 10x10 problem.

Here we test the effect of product time modification on the problem from Test 5.

Results

The start evaluation with $\alpha = 1$ is 699 which is only slightly better than 655 which is the lower bound given by f_1 ($\alpha = 0$). Recall that the optimum is 930 and that the resource based lower bound from f_2 was 631. This suggests that the difficulty of this problem lies not in a resource bottleneck or a certain critical product. The results are shown in Table 7.11.

Method	α	$l = 0$	$l = 2$	$l = 5$	$l = 10$	$l = 20$	$l = 50$	$l = \infty$
Best Enabled	0	1092	1095	1099	1118	1121	1526	1526
Best Enabled	0.25	1173	1223	1223	1201	1267	1689	1689
Best Enabled	0.5	1199	1153	1166	1179	1165	1526	1526
Best Enabled	0.75	1120	1153	1170	1105	1139	1443	1443
Best Enabled	1	1168	1140	1101	1131	1156	1180	1180
Rand. Best	0	1046	1047.8	1040.6	1055.9	1094.2	1342.0	1475.8
Rand. Best	0.25	1120	1218	1223	1169	1130	1614.1	1622.1
Rand. Best	0.5	1142	1123	1114	1143	1147	1515.4	1517
Rand. Best	0.75	1120	1123	1170	1098	1108.8	1439.5	1430.1
Rand. Best	1	1055	1056	1101	1131	1194	1165.3	1164.0
DFS1	0	1092	1094	1094	1118	1097	1526	1526
DFS1	0.25	1173	1181	1181	1181	1206	1631	1631
DFS1	0.5	1199	1106	1114	1126	1165	1517	1517
DFS1	0.75	1120	1106	1135	1105	1110	1417	1417
DFS1	1	1168	1140	1101	1131	1156	1180	1180
Restart1	0	1046	1059	1099	1071	1095	1410	1526
Restart1	0.25	1099	1095	1084	1099	1098	1208	1208
Restart1	0.5	1073	1089	1085	1100	1137	1272	1272
Restart1	0.75	1063	1078	1087	1105	1085	1204	1204
Restart1	1	1093	1104	1101	1091	1090	1128	1154
DFS2	0	1087	1092	1094	1118	1097	1450	1486
DFS2	0.25	1123	1142	1174	1181	1181	1537	1591
DFS2	0.5	1149	1072	1077	1126	1165	1487	1491
DFS2	0.75	1120	1091	1112	1087	1110	1413	1417
DFS2	1	1144	1140	1061	1131	1156	1172	1172
Restart2	0	1045	1047	1058	1073	1107	1435	1526
Restart2	0.25	1053	1063	1084	1069	1098	1208	1217
Restart2	0.5	1040	1063	1085	1096	1137	1272	1272
Restart2	0.75	1063	1063	1087	1105	1139	1155	1155
Restart2	1	1033	1063	1101	1092	1136	1128	1180
Beam	0	1023	1030	1023	1050	1011	1149	1169
Beam	0.25	985	1004	1025	991	1053	1105	1120
Beam	0.5	985	993	985	988	1034	1109	1161
Beam	0.75	985	991	985	988	1030	1050	1114
Beam	1	985	994	979	985	1030	1105	1152
Limited A*	0	1020	1065	1065	1045	1153	1482	1453
Limited A*	0.25	1051	1003	1107	1059	1118	1462	1617
Limited A*	0.5	1092	1065	1087	1097	1113	1436	1551
Limited A*	0.75	1072	1062	1057	1075	1159	1556	1244
Limited A*	1	1027	1074	1043	1136	1104	1400	1537

Table 7.11: Results for Test 10.

Additional test facts

- *Rand. Best.* For $\alpha > 0$, the algorithm has time for approximately 390 trials for $l = 0$ and 100 trials with $l = \infty$. The standard deviation is zero in many cases, due to the fact that when deciding the next transition to fire, the modified evaluation function does not give many alternatives that have the same integer value.
- *Beam.* The beam width that gives a running time near the limit is around 240 for $l = 0$ and 80 for $l = \infty$.
- *Limited A*.* Values between 19 (high lookaheads) and 45 (small lookaheads) for u give a running time near the time limit.

Discussion

The impact of the modification parameter α is different from the previous test. Beam Search improves its results with higher α. For the other algorithms, there is no clear trend, except for Randomized Best Enabled which gets worse results with $\alpha > 0$ than without modification.

7.4.6 Conclusions

The evaluation function f_4 based on modified remaining product times cannot be said to work better for the makespan objective than the admissible evaluation functions tested earlier, although it in some cases finds better results. Together with Beam Search it gives acceptable results, and we can draw the important conclusion that this combination does not show any unwanted bad behaviour. Another observation that suggests that the modification procedure is sound is that the start estimation for $\alpha = 1$ is realistic in all cases but for the difficult 10x10 problem.

7.5 Tests with tardiness penalties

Now we come to the most interesting test problems, namely the ones where we have minimization of tardiness penalties as the objective. Our results from the much simpler case of makespan objective lead us to believe that Beam Search should be the best of the informed search methods, and the most important question is perhaps how this method with evaluation function f_5 and $\alpha > 0$ will compare with $\alpha = 0$ and with the uninformed search method Randomized Firing. With high values of α (> 0.5), we can expect f_5 to overestimate the objective value heavily in some cases, since it assumes that all products are given low priority. In spite of this, we believe that it can still work well when comparing different search nodes.

We note that the objective value of different schedules varies much more for tardiness penalties than for makespan, especially if the magnitude of penalties differ between product types. In the ideal case there are no late products and the objective value is then zero. But in the case of a late product with a high penalty, we will get a very large objective value. Therefore it is not very meaningful to analyze the quotient between the objective values of two different schedules in the same way we would do for e.g. makespan. In the total costs case (which is the long-term goal of this research project) such a comparison is more relevant, since then the tardiness penalties will be just one part of the total objective value. We urge the reader to keep this in mind when studying the following results.

7.5.1 Test 11. A small number of products.

For our first test on tardiness penalties, we take a problem that should not be too difficult to solve to optimality. We have one product of each type in the work center example from Section 7.1.1. For simplicity, we assume linear tardiness penalties. The due date and the penalty factor (per unit of time above the due date) for the different product types are shown in Table 7.12. Since the problem is rather small and we do not have a high workload on the resources, the admissible evaluation function ($\alpha = 0$) can be expected to do well. The aim of this test is to see how Beam Search with the non-admissible evaluation function and Randomized Firing will do compared to optimum.

Results

The start evaluation for α up to about 0.15 is 0. For $\alpha = 1$ it is 17. The optimum objective value is 39, a solution which can be found for $l \geq 4$. For $l = 3$ the optimum is 51 and for $l \leq 2$ it is 54. The results within the time limit are shown in Table 7.13.

Product type	Due date	Penalty factor
P1	60	2
P2	80	1
P3	70	4
P4	90	3

Table 7.12: Data for Test 11.

Additional test facts

- *Randomized.* About 21250 trials can be done within the time limit. The standard deviation is zero for small lookaheads and around 2 for higher.
- *Rand. Best.* The algorithm has time for approximately 2670 trials for $l = 0$ and 1920 trials with $l = \infty$. The results are without deviation in most cases, and the standard deviation is below 2 in all cases.
- *DFS2/Restart2.* It takes around 30/19 CPU seconds for DFS2/Restart2 respectively to prove 39 optimal for $l = \infty$ (using $\alpha = 0$).
- *Beam.* A beam width of 80 is enough in most cases for finding the optimum, which corresponds to running times of up to about 0.5 CPU seconds (for $l = \infty$).

Discussion

This test indicates that tardiness penalties problems are more difficult than makespan problems, and that a slightly longer lookahead is needed for finding the optimum. Although the search tree for this problem is of the same size as the one in Test 1, we have much larger variations in objective value for the different methods. Beam Search and Limited Expansion A* Search work well for all values of α. The branch-and-bound methods with duplicate checking do well in most cases, but there are also some bad values, especially for high values of α. An interesting fact is that most methods using duplicate checking miss the optimum at $l = 3$ (Limited Expansion A* Search finds it at $\alpha = 0.75$). The possibility of this was discussed in Section 6.1.6.

7.5.2 Test 12. An average number of products.

In this test problem, we again use our work center example from Section 7.1.1 as the basis. We have three items of each product type, giving a total of 12 products. As in the previous case, we have linear tardiness penalties. The due dates and penalty factors for the different product types are shown in Table 7.14.

Results

The start evaluation for α up to about 0.65 is 0. For $\alpha = 0.75$ it is 57 and for $\alpha = 1$ it is as high as 354. We note here that the best results known for the problem are 139 for $l = 0$, and 114 for $l \geq 1$. The results within the time limit are shown in Table 7.15.

Method	α	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=10$	$l=\infty$
First Enabled	-	133							
Randomized	-	54	54	54	51.9	47.2	47.3	47.3	50.7
Best Enabled	0	133	133	133	133	133	133	133	133
Best Enabled	0.25	92	92	92	92	96	97	92	71
Best Enabled	0.5	92	92	92	92	96	97	92	71
Best Enabled	0.75	92	92	92	92	96	97	92	92
Best Enabled	1.0	218	218	218	218	218	218	218	218
Rand. Best	0	54	54	54	54	54	54	47.7	49.8
Rand. Best	0.25	54	54	54	54	54	54	54	54
Rand. Best	0.5	54	54	54	54	54	54	54	54
Rand. Best	0.75	54	54	54	54	54	54	54	54
Rand. Best	1.0	122	122	122	129	171	171	175	183
DFS1	0	94	94	94	51	47	47	47	65
DFS1	0.25	54	54	54	54	96	97	92	56
DFS1	0.5	54	54	54	54	96	97	92	54
DFS1	0.75	63	63	63	54	51	54	54	56
DFS1	1.0	54	63	63	63	63	63	54	54
Restart1	0	54	63	63	63	63	63	47	47
Restart1	0.25	54	63	63	63	63	63	47	47
Restart1	0.5	54	55	55	71	96	97	47	47
Restart1	0.75	63	54	54	54	54	54	51	51
Restart1	1.0	55	55	55	55	52	56	51	51
DFS2	0	54	54	54	54	39	39	39	39
DFS2	0.25	54	54	54	54	39	39	39	47
DFS2	0.5	54	54	54	54	39	39	39	39
DFS2	0.75	63	63	63	63	39	39	39	39
DFS2	1.0	54	63	63	63	39	39	39	39
Restart2	0	54	54	54	54	39	39	39	39
Restart2	0.25	54	54	54	54	39	39	39	39
Restart2	0.5	54	54	54	54	39	39	39	39
Restart2	0.75	63	54	54	54	39	39	39	39
Restart2	1.0	55	55	55	55	39	39	39	39
Beam	0	54	54	54	54	39	39	39	39
Beam	0.25	54	54	54	54	39	39	39	39
Beam	0.5	54	54	54	54	39	39	39	39
Beam	0.75	54	54	54	54	39	39	39	39
Beam	1.0	54	54	54	54	39	39	39	39
Limited A*	0	54	54	54	54	39	39	39	39
Limited A*	0.25	54	54	54	54	39	39	39	39
Limited A*	0.5	54	54	54	54	39	39	39	39
Limited A*	0.75	54	54	54	51	39	39	39	39
Limited A*	1.0	54	54	54	54	39	39	39	39

Table 7.13: Results for Test 11.

Additional test facts

- *Randomized.* Around 7250 trials are made within the time limit. The standard deviation increases from around 12 for $l = 0$ up to 85 for $l = \infty$.

- *Rand. Best.* The algorithm has time for approximately 590 trials for $l = 0$ and 390 trials with $l = \infty$. The standard deviation is between 20 and 40 in most cases.

- *Beam.* The beam width that gives a running time near the limit is around 460 for $l = 0$ and 220 for $l = \infty$. Since the best results were obtained with $\alpha = 1$ we also tried increasing α beyond 1, but this made the results worse.

- *Limited A*.* The value of u that gives a running time near the limit is between 40 and 60 in most cases.

Product type	Due date	Penalty factor
P1	150	2
P2	200	1
P3	180	4
P4	220	3

Table 7.14: Data for Test 12.

Discussion

We see that the results have a huge variation, and by examining the schedules we can see that ones with similar makespan can have totally different objective value. All methods except Randomized Best Enabled are in general helped by modified remaining product time calculations ($\alpha > 0$). Beam Search with $\alpha = 1$ gives the best results by a large margin, and this confirms our belief that high values of the modification parameter should be good for tardiness penalties. Randomized Firing does pretty well for small lookaheads. The Randomized Best Enabled method does not give very good results except perhaps low values of α and l. The branch-and-bound methods give disastrous results except for a few lucky shots. Limited Expansion A* Search gives a few fairly good values but also many bad ones. These results show that minimizing tardiness penalties can be a very difficult problem.

7.5.3 Test 13. A large number of products.

We now test an even larger problem to see if we get similar results. We have now five items of each product type, giving a total of 20 products. As in the previous problems, we assume linear tardiness penalties. The due date and the penalty factor for the different product types are shown in Table 7.16.

Results

The start evaluation is 0 for α up to about 0.75. For $\alpha = 1$ it is 750. The best schedule known for the problem has a value of 201 (found with $l = 0$). The results are shown in Table 7.17.

Additional test facts

- *Randomized.* About 4430 trials are made within the time limit. The standard deviation lies between 20 and 60, except for $l = \infty$ where it is over 100.
- *Rand. Best.* The algorithm has time for approximately 280 trials for $l = 0$ and 195 trials with $l = \infty$. The standard deviation is between 20 and 80 except for some cases with large lookahead.
- *Beam.* The beam width that gives a running time near the limit is around 280 for $l = 0$ and 130 for $l = \infty$.
- *Limited A*.* The values of u that give a running time near the limit are between 20 and 40 in most cases.

Method	α	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$	$l = 10$	$l = \infty$
First Enabled	-	718							
Randomized	-	186.2	221.7	199.4	213.1	238.9	261.9	358.9	539.1
Best Enabled	0	718	730	754	799	844	844	844	844
Best Enabled	0.25	706	706	830	821	821	920	920	920
Best Enabled	0.5	630	669	669	669	721	695	1125	1125
Best Enabled	0.75	597	609	609	623	435	435	758	758
Best Enabled	1.0	1093	1012	1012	1012	1114	1205	1564	1418
Rand. Best	0	221.3	250.6	241.8	254.5	284.6	300.6	375.1	447.7
Rand. Best	0.25	212.8	271.8	262.4	283.8	305.8	320.7	343.0	417.4
Rand. Best	0.5	245.5	249.0	272.3	255.3	268.9	292.4	299.0	342.2
Rand. Best	0.75	406.7	375.5	405.3	385.8	398.9	378.9	344.6	362.6
Rand. Best	1.0	521.9	571.4	591.0	560.2	560.3	555.7	571.2	582.5
DFS1	0	664	676	700	700	700	719	719	719
DFS1	0.25	640	655	797	670	676	866	866	866
DFS1	0.5	555	621	621	481	637	546	1004	937
DFS1	0.75	271	271	271	271	288	419	526	526
DFS1	1.0	334	773	773	842	580	778	971	1078
Restart1	0	528	528	528	550	550	545	545	603
Restart1	0.25	382	487	495	515	515	515	463	463
Restart1	0.5	195	195	234	219	253	279	443	443
Restart1	0.75	545	557	557	393	352	352	490	676
Restart1	1.0	275	304	298	301	301	301	346	495
DFS2	0	328	443	582	652	646	646	646	595
DFS2	0.25	328	333	515	413	596	800	734	667
DFS2	0.5	192	256	306	306	474	415	873	732
DFS2	0.75	266	266	266	266	273	275	440	420
DFS2	1.0	334	408	257	329	602	753	791	819
Restart2	0	456	555	550	550	550	594	628	628
Restart2	0.25	456	505	514	553	618	660	528	528
Restart2	0.5	195	195	517	390	390	510	463	523
Restart2	0.75	397	310	376	382	346	346	551	672
Restart2	1.0	316	302	283	318	318	308	304	498
Beam	0	146	194	167	192	419	313	299	219
Beam	0.25	152	154	154	184	229	303	282	193
Beam	0.5	166	166	232	233	244	232	187	290
Beam	0.75	139	139	150	150	148	157	150	160
Beam	1.0	150	118	128	120	128	127	129	142
Limited A*	0	285	343	154	209	237	202	209	337
Limited A*	0.25	294	244	449	246	199	208	202	373
Limited A*	0.5	285	238	258	228	362	292	252	256
Limited A*	0.75	258	194	178	266	262	198	240	210
Limited A*	1.0	190	190	207	201	628	464	183	317

Table 7.15: Results for Test 12.

Discussion

This test amplifies many of the trends seen in the previous one. All methods except Randomized Best Enabled improve their results when modified remaining product time calculations are used ($\alpha > 0$). The branch-and-bound methods give their best results (which still are very bad) at $\alpha = 1$, whereas the best setting seems to be $\alpha = 0.75$ for Beam Search and Limited Expansion A* Search. Beam Search again gives the best results. Randomized Firing does well for $l = 0$.

7.5.4 Conclusions

These tests on tardiness penalties show the merit of the modified remaining product time calculations. They also support the conclusions from the makespan tests that Beam Search gives the best results and that Randomized Firing with a small lookahead (zero for the larger problems) is a fairly good and stable alternative to the former. The best results are obtained with high values of the modification parameter ($\alpha > 0.5$) , in most cases with the worst case estimation $\alpha = 1$. This confirms our

Product type	Due date	Penalty factor
P1	350	3
P2	400	2
P3	250	1
P4	300	4

Table 7.16: Data for Test 13.

theoretical discussions.

Method	α	$l = 0$	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$	$l = 10$	$l = \infty$
First Enabled	-	3608							
Randomized	-	347.7	449.6	488.0	538.4	588.0	703.8	1054.1	1708.9
Best Enabled	0	3631	3631	3651	3663	3663	3663	3663	3663
Best Enabled	0.25	3608	3907	3953	4005	4005	4005	4005	4017
Best Enabled	0.5	3493	3677	3677	3677	3701	3701	3701	3701
Best Enabled	0.75	2412	3232	3232	3232	3232	3855	4199	4199
Best Enabled	1.0	1970	1941	1941	1717	2121	1789	2899	3325
Rand. Best	0	452.9	538.9	566.0	587.2	662.7	727.5	849.3	1190.2
Rand. Best	0.25	474.4	567.9	591.2	628.8	682.7	752.7	908.9	1076.4
Rand. Best	0.5	519.6	572.4	560.1	625.4	696.9	760.8	870.8	935.0
Rand. Best	0.75	629.3	656.5	640.1	713.8	732.0	758.4	859.8	948.4
Rand. Best	1.0	823.8	837.7	827.8	849.3	834.6	832.1	917.3	1047.9
DFS1	0	3631	3631	3651	3651	3651	3651	3651	3651
DFS1	0.25	3608	3907	3953	3953	3953	3953	3953	3953
DFS1	0.5	3493	3677	3677	3677	3701	3701	3701	3701
DFS1	0.75	1863	2853	2956	2700	2952	3668	4165	3907
DFS1	1.0	1162	1323	1341	1285	1202	1449	2737	2987
Restart1	0	2493	2493	2493	2528	2588	2588	2613	2626
Restart1	0.25	2224	2294	2302	2294	2366	2366	2366	2366
Restart1	0.5	2224	2294	2294	2310	2334	2334	2334	2334
Restart1	0.75	1483	1351	1761	1805	1955	1955	2517	2769
Restart1	1.0	1399	1752	1776	1717	1417	1789	2514	2665
DFS2	0	3027	3308	3381	3483	3570	3570	3570	3631
DFS2	0.25	2904	3607	3708	3822	3892	3934	3934	3829
DFS2	0.5	2366	2578	3137	3438	3586	3627	3677	3485
DFS2	0.75	1175	1708	1955	2704	2895	3696	3737	3649
DFS2	1.0	1162	1197	1309	1285	1249	995	2561	2965
Restart2	0	2493	2493	2493	2553	2626	2626	3064	3122
Restart2	0.25	2224	2294	2302	2294	2366	2366	3098	3110
Restart2	0.5	2224	2294	2294	2310	2334	2334	2571	2822
Restart2	0.75	1483	1809	1941	1955	2374	2735	2769	3326
Restart2	1.0	1271	731	1591	1499	1417	1745	1545	2213
Beam	0	1111	750	1020	1012	1444	1434	1404	1428
Beam	0.25	631	531	697	837	861	816	937	681
Beam	0.5	319	435	499	473	565	557	451	508
Beam	0.75	201	229	499	441	545	534	1016	726
Beam	1.0	336	380	394	469	531	577	737	532
Limited A*	0	1430	1402	1382	1813	1756	1512	1954	1494
Limited A*	0.25	940	844	973	1299	884	641	980	1386
Limited A*	0.5	1003	523	597	677	509	711	533	749
Limited A*	0.75	512	571	445	341	730	872	751	782
Limited A*	1.0	515	773	736	787	∞	898	1185	1174

Table 7.17: Results for Test 13.

Chapter 8

Summary and conclusions

Now we will summarize the main results from this work and draw some general conclusions.

8.1 General remarks on modelling and solution techniques

We have studied an application that requires the challenging combination of detailed modelling and short execution time. The decision support system is intended to compare a number of possible scenarios when consulted by a user. We have identified the modelling and solution of the scheduling problems associated to these scenarios as the key to good performance for the system. After studying techniques for similar scheduling problems, we come to the conclusion that there is typically a conflict between modelling power and solution efficiency. The most efficient solution methods are based on the disjunctive graph model, but they are hard to generalize to more realistic models. CLP and discrete event methods offer the best modelling features, but have different types of efficiency problems.

Although CLP schedulers have fast specialized propagation mechanisms for some of the basic scheduling constraints, they have to rely on more general and possibly slower techniques for other constraints. Some modelling features are possible to include in the scheduling system, but require several constraints and complex expressions. This raises some question marks regarding a CLP scheduler's ability to produce solutions fast also for larger problems.

Discrete event based schedulers face another kind of problem: they are restricted to use sequential search methods. The search tree to explore has a very high degree of redundancy with respect to different paths, making it impossible to solve larger problems optimally. However, finding some solution fast is very easy, so properly designed search methods will always give a decent schedule even for large problems.

8.2 The Petri net model

For reasons discussed, we chose a discrete event model for the scheduling system. We decided to use a Petri net as the basis for this model, the main reasons for this are:

- It has a graphical representation.

- Its behaviour is clearly defined and can easily be expressed mathematically.
- It allows for many of the modelling features we required.
- Research done on Petri net scheduling shows some promising results.

By using a partially new timing concept with timestamps for tokens and time delays associated to output arcs we could further increase the modelling power, at the expense of a more complicated and memory-consuming implementation giving a slightly decreased speed. It is possible to take further steps in this direction, by allowing some tokens to carry more information and to have additional data associated to transitions and arcs.

8.3 Scheduling problems

We have mainly studied two problem types, minimization of makespan and minimization of tardiness penalties, with the aim of dealing with a third: minimization of total costs. Because of its simplicity the makespan problem has been extensively researched using different types of models and solution methods, but its practical usefulness is limited. Still, it remains one of the more challenging problems in combinatorial optimization. However, compared to the minimization of tardiness penalties, makespan problems are pretty well behaved.

Not only is the former problem harder to model, it is also much more difficult to solve. Apart from making sure that the overall production rate is high in the system, the order in which products are completed is crucial. One effect of this is that instances of the problem show large variations in objective value between schedules that are of similar makespan. It is also hard to find good estimates of the objective value for partial schedules, especially if one desires a lower bound, and this makes it difficult to build informed search methods that work well.

Luckily, the step from minimization of tardiness penalties to minimization of total costs is not that long, or at least so we believe. Adding costs for activities should not introduce any new problems, it is just the book-keeping that gets slightly more complicated.

8.4 Solution methods

We have evaluated a number of solution methods on a set of test problems, with different objectives, evaluation functions and parameter settings. We can come to a conclusion regarding a feature that all solution methods share: the limited lookahead heuristic is a good addition to all of them. Without imposing any computational overhead, it makes a reduction of the search space, the magnitude of which is controllable with the lookahead parameter l. With an appropriate choice of the latter, there are still many good paths in the reachability graph even though its size is significantly smaller than without the heuristic. This proves to be beneficial for time-limited algorithms, especially for large problems.

Even though we are mostly interested in minimization of tardiness penalties, our general conclusions on the merit of different solution methods hold also for the makespan objective. Beam Search shows the overall best results. The success is probably due to its ability to carry several partial solutions in parallel until it is visible in the evaluation function which ones are the best. This means that it works well also when the evaluation function gives guidance globally but not so much locally (i.e. selecting the next transition to fire for a given marking), whereas this is not the case for the depth-oriented search methods. The drawback of Beam Search is that it does not give a complete schedule if interrupted prematurely, so one has to let it run until it is done. Fortunately, the execution time is pretty easy to predict and control by the setting of the beam width. Using some empirical data, it should be possible to construct the system so that the problem size is analysed and the beam width adjusted prior to execution, giving a resulting running time quite close to the desired one. If one is willing to sacrifice the fixed time limit, an alternative is to use a fixed beam width for all problem instances. This will give better results for the larger problems at the expense of an increased running time. The tests show that the results become fairly stable at a good level around a beam width of $150 - 200$.

Apart from Beam Search, we recommend that the Randomized Firing algorithm is implemented into the system. This recommendation is based mainly on its great flexibility to be used with any modelling feature and objective function, but also the fact that it for small lookaheads gives fairly good and stable results for all problems tested. For larger problems, the lookahead should probably be set to zero (i.e. eager firing) in the algorithm. An alternative to using only one of these methods for a particular problem is to use a hybrid method, allocating e.g. 25 % of its time on Randomized Firing and the rest on Beam Search.

The rest of the tested solution methods are for different reasons not as suitable for the scheduling system, at least not in their present form. The Randomized Best Enabled method does not have the flexibility of the uninformed Randomized Firing algorithm, so the only reason for including it in the system would be if it could be improved so that it would outperform Beam Search. A possible development of the method would be to extend the random selection to choices with similar but not necessarily exactly the same evaluation. Although this would probably eliminate some of the bad behaviour observed in the tests, it does not seem very likely that this would give a significant overall improvement. The branch-and-bound algorithms give bad results for the larger problems, so they are not useful other than for optimality proofs of smaller problems. The Limited Expansion A* method is not as efficient as Beam Search, and has in addition some unwanted characteristics. It occasionally gives quite bad results, and even worse, it sometimes fails to produce any solution. The running time is also much harder to estimate a priori.

We have seen for the makespan problems that it is advantageous to use an evaluation function that focuses both on products and resources. For the tardiness penalties problems, it is not only advantageous but absolutely necessary for larger problems. The approximate evaluation function f_5 is one of the most important contributions of this work. It modifies the remaining processing time of a product according to the workload on the resources used, giving a much more useful estimate than the lower bound obtained without modification. In agreement with the theoretical discussion, the best values for the modification parameter α proved to be the ones corresponding to low product priority ($\alpha > 0.5$) for the tests made with beam search on tardiness penalties.

8.5 Limitations

The model and solution methods we have developed are intended for scheduling problems with many detailed real-world constraints that must be solved with respect to total costs within a short limited computing time. How useful is our approach when we change some of these conditions?

We start with relaxing the limited computation time, and allow longer times (say a few minutes) that can vary between different problem instances. The randomized algorithms will probably only improve their results marginally given more computing time, whereas the other algorithms can be expected to improve more. On large-size problems, the branch-and-bound methods will probably still not be able to compete with the other methods. We believe that Beam Search would still perform best. However, previous approaches in Petri net scheduling, such as a pure A* algorithm with a depth-dependent evaluation function, may also be useful in such a case. It would also be interesting to compare the performance with a CLP-approach having the same modelling features.

Our scheduling model is designed for problems that are not extremely large, say up to about 10 machines and 30 jobs. If used for larger problems, we would probably have to increase running times in order to get reasonably good results. But for such a large-scale problem it is probably better to solve a simplified problem on a general level, taking care of details in more manageable sub-problems.

If we relax many of the modelling requirements we have for our system so that e.g. disjunctive graph or Lagrangian relaxation approaches also could be used for the problem, they would probably outperform our methods, at least if we allowed slightly more computing time.

8.6 New contributions

To summarize, we will now point out some areas where we have made new contributions compared with previous research.

- The application.
    - The idea of a decision support system for scheduling on the detailed work center level is to the best of our knowledge new.
- The literature review.
    - We studied existing modelling and solution techniques from different research areas with the aim of investigating the possibilities for quickly solving a detailed scheduling problem with total costs as the objective.
- The Petri net model.
    - The new timing concept allowed us to build a model more general and compact than previous approaches in Petri net scheduling.
    - We created a total costs objective function in the Petri net model.
- Time-limited solution methods for Petri net scheduling.
    - We evaluated several solution methods suitable for a short limited computing time. As far as we know, the only methods among those we tried that have been tested earlier are branch-and-bound with depth-first-search and the limited expansion A* search algorithm.
    - We discovered that global beam search can produce very good results.

- Evaluation functions.
  - We created building blocks for evaluation functions that allow for several modelling features.
  - We developed the evaluation function f_5 for the objective of minimizing tardiness penalties. This function has the ability to take the workload of resources into account when estimating the remaining time until a product is completed, which is necessary when the number of products is large. The function proved to work well with beam search in tests.
- Search space reduction techniques.
  - We introduced the limited lookahead heuristic which improves the efficiency of most methods.
  - We examined how search space reduction techniques affect optimality. We discussed the effect of limited lookahead (and in particular non-delay scheduling) and duplicate checking and the combination of the two.

Chapter 9

Suggestions for future research and development

In this chapter we will point out some possible directions of future research and development for the decision support system.

9.1 Further tests with the current algorithms

It would be useful if the algorithms were tested on other example problems, in order to confirm or reject the conclusions drawn so far. The new tests should be focused mainly on minimization of tardiness penalties. Since the examples tested here only include a subset of the possible modelling features, a couple of new work center examples having some more could be tested. The most important feature not yet tested is flexible routing.

9.2 Improving the algorithms

The limited lookahead heuristic and duplicate checking are two mechanisms that reduce the search space. One line of research could be to investigate if it is advantageous to incorporate other search space reduction techniques into the algorithms. These techniques could be either exact (i.e. guaranteed to not cut off all optimal paths) like the duplicate checking[1], or approximate like the limited lookahead heuristic. One example could be to always fire transitions that are not in conflict with any other enabled transition as soon as possible.

9.3 Analyzing the modified remaining product time calculations

The modified remaining product time calculations have shown very good results when used with beam search for the minimization of tardiness penalties. We have given some arguments for the soundness of these calculations and discussed its physical interpretation, but we are still a bit away from understanding its actual behaviour

[1]For full lookahead.

during a search process. Studying the calculations in detail for some search nodes of a problem might give more insight to e.g. why it is not always better than unmodified calculations at the local level of evaluating child nodes for the next transition to fire. This could give clues of how to further improve the evaluation function.

We note that we have during tests found some interesting facts that lead us to believe that it can be worthwhile to investigate these calculations further. In an early implementation of f_5, the correct timestamps of tokens arriving earlier than the due date to goal places were replaced by the due dates. This does of course not affect the objective value, but it gives $t^{PM}_{\text{avg}}(M)$ another value than intended. Interestingly, the correct implementation of f_5 (for which we have presented results) does not always give better results than this first (erroneous) version. For example, the best value (114) known for the problem of Test 12 was found with the first version. This indicates that the time which $t^R(M, R)$ is compared with in (6.2) is important for the performance. One idea for investigating this could be to replace $t^{PM}_{\text{avg}}(M)$ by $\beta t^{PM}_{\text{avg}}(M)$ in (6.2), and test different values for the parameter β.

9.4 Developing a total costs model

The most natural continuation of this project is of course to extend the methods for minimization of tardiness penalties to the total costs case, a task that should not be too difficult. When this is done, one can start to test the system in decision support situations with multiple scheduling scenarios. When the solution methods work satisfactorily for these problems, focus can be shifted towards improving the implementation of the test versions in order to reduce running time and memory consumption. As mentioned earlier, it should be possible to improve the efficiency considerably.

9.5 Improving the decision support logic

Instead of a priori deciding which scheduling scenarios to evaluate, one could investigate strategies that add certain scenarios only if other scenarios have been investigated first and found not to be satisfactory. For example, if a scenario leads to a schedule where all due dates are met, it is of course a waste of time to also investigate a scenario with the same starting conditions but with overtime ordered. One could also analyze if it could be worthwhile to examine other scenarios than those stated in Section 5.4.6, e.g. buying a product type that shows to be difficult to deliver in time or hiring an additional operator if this resource is identified as a bottleneck.

Another way of improving the system could be to investigate scenarios that correspond to more than one special action. There must be some intelligence in the system for selecting only the relevant combinations of actions, otherwise the number of scenarios may be too large.

9.6 Trying other approaches

It would also be of interest to further investigate if we could get the same modelling features as for the Petri net based scheduler with another approach. And if this shows to be the case, we should of course try to compare the methods quantitatively on some test problems. For most approaches this would require large amounts of work, but a candidate that would be more easy to evaluate is the CLP-based commercial product ILOG SCHEDULE which has many of the modelling features we are looking for.

Bibliography

E. H. L. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines.* Wiley, Chichester, 1989.

J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401, 1988.

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

D. Azzopardi and S. Lloyd. Reduction of search space for scheduling of multi-product batch process plant through Petri net modelling. In *Proceedings of the 2nd IFAC/IFIP/IFORS Workshop on Intelligent Manufacturing Systems*, pages 371–376, 1994.

P. Baptiste and C. Le Pape. Disjunctive constraints for manufacturing scheduling: Principles and extensions. In *Proceedings of the Third International Conference on Computer Integrated Manufacturing*, 1995.

P. Baptiste, C. Le Pape, and W. Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. In *Proceedings of the First International Joint Workshop on Artificial Intelligence and Operations Research*, 1995.

J. W. Barnes, M. Laguna, and F. Glover. An overview of tabu search approaches to production scheduling problems. In D. E. Brown and W. T. Scherer, editors, *Intelligent Scheduling Systems*, pages 101–127. Kluwer Academic Publishers, Boston, 1995.

J. E. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

I. Ben Abdallah, H. ElMaraghy, and T. ElMekkawy. An efficient search algorithm for deadlock-free scheduling in FMS using Petri nets. In *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, pages 1793–1798, 1998.

J. R. Birge and F. Louveaux. *Introduction to stochastic programming.* Springer-Verlag, New York, 1997.

J. Błażewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Węglarz. *Scheduling Computer and Manufacturing Processes.* Springer-Verlag, Berlin, 1996.

D. E. Brown and W. T. Scherer, editors. *Intelligent Scheduling Systems.* Kluwer Academic Publishers, Boston, 1995.

P. Brucker. *Scheduling algorithms.* Springer-Verlag, Berlin, second edition, 1998.

A. Camurri, P. Franchi, F. Gandolfo, and R. Zaccaria. Petri net based process scheduling: A model of the control system of flexible manufacturing systems. *Journal of Intelligent and Robotic Systems*, 8:99–123, 1993.

Y. Caseau and F. Laburthe. Disjunctive scheduling with task intervals. Technical report, LIENS Technical Report 95-25, École Normale Supérieure Paris, France, 1995.

O. V. K. Chetty and O. C. Gnanasekaran. Modelling, simulation and scheduling of flexible assembly systems with coloured Petri nets. *International Journal of Advanced Manufacturing Technology*, 11(6):430–438, 1996.

A. K. Chincholkar and O. V. K. Chetty. Stochastic coloured petri nets for modelling and evaluation, and heuristic rule base for scheduling of FMS. *International Journal of Advanced Manufacturing Technology*, 12:339–348, 1996.

P. Chrétienne, E. G. Coffman, J. K. Lenstra, and Z. Liu, editors. *Scheduling theory and its applications.* John Wiley & Sons Ltd., Chichester, 1995.

F. DiCesare, G. Harhalakis, J. Proth, J. Silva, and F. Vernadat. *Practice of Petri nets in manufacturing.* Chapman & Hall, London, 1993.

U. Dorndorf and E. Pesch. Evolution based learning in a job shop scheduling environment. *Computers and Operations Research*, 22:25–40, 1995.

H. Fisher and G. L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In J. F. Muth and G. L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice Hall, Englewood Cliffs, New Jersey, 1963.

T. Frühwirth, A. Herold, V. Küchenhoff, T. L. Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint Logic Programming - an informal introduction. Technical report, ECRC-93-5, European Computer-Industry Research Centre, Germany, 1993.

M. R. Garey and D. S. Johnson. *Computers and intractability.* W. H. Freeman and Co., San Francisco, Calif., 1979.

F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, Norwell, USA, 1997.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, Mass., 1989.

M. Gondran and M. Minoux. *Graphs and algorithms.* John Wiley & Sons Ltd., Chichester, 1984.

R. Haupt. A survey of priority rule-based scheduling. *OR Spektrum*, 11(1):3–16, 1989.

W. S. Herroelen and E. L. Demeulemeester. Recent advances in branch-and-bound procedures for resource-constrained project scheduling problems. In P. Chrétienne, E. G. Coffman, J. K. Lenstra, and Z. Liu, editors, *Scheduling theory and its applications*, pages 259–276. John Wiley & Sons Ltd., Chichester, 1995.

J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the theory of neural computation.* Addison Wesley, Massachusetts, USA, 1991.

D. J. Hoitomt, P. B. Luh, and K. R. Pattipati. A practical approach to job-shop scheduling problems. *IEEE Transactions on Robotics and Automation*, 9(1):1–13, 1993.

O. Holthaus and H. Ziegler. Look ahead job demanding for improving job shop performance. *OR Spektrum*, 19(1):23–29, 1997.

M. D. Jeng, S. C. Chen, and L. C. Shi. A search approach based on the Petri net theory for FMS scheduling. In *Preprints of the Thirteenth IFAC World Congress*, pages 55–60, 1996.

M. D. Jeng, R. W. Jaw, and P. L. Hung. Scheduling FMS with due dates based on Petri net state equations. In *Proceedings of the 1997 IEEE International Conference on Systems, Man, and Cybernetics*, number 3, pages 2724–2729, 1997.

M. D. Jeng and C. S. Lin. Scheduling flexible manufacturing systens containing assembly operations based on Petri net structures and dynamics. In *Proceedings of the 1997 IEEE International Conference on Systems, Man, and Cybernetics*, number 5, pages 4439–4435, 1997.

K. Jensen. A brief introduction to coloured Petri nets. *Lecture Notes in Computer Science*, 1217:203–208, 1997.

V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

C.-Y. Lee, L. Lei, and M. Pinedo. Current trends in deterministic scheduling. *Annals of Operations Research*, 70:1–41, 1997.

D. Y. Lee and F. DiCesare. Scheduling FMS using Petri nets and heuristic search. *IEEE Transactions on Robotics and Automation*, 10(2):123–132, 1994.

T. Liljenvall. Scheduling for production systems. Technical report, No. 293L, Department of Signals and Systems, Control Engineering Laboratory, School of Electrical and Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1998.

P. B. Luh, L. Gou, Y. H. Zhang, T. Nagahora, M. Tsuji, K. Yoneda, T. Hasegawa, Y. Kyoya, and T. Kano. Job shop scheduling with group-dependent setups, finite buffers, and long time horizon. *Annals of Operations Research*, 76:233–259, 1998a.

P. B. Luh, X. Zhao, Y. Wang, and L. S. Thakur. Lagrangian relaxation neural networks for job shop scheduling. In *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, pages 1799–1804, 1998b.

R. J. Mayer, C. P. Menzel, M. K. Painter, P. S. deWitte, T. Blinn, and B. Perakath. Information integration for concurrent engineering (IICE): IDEF3 process description capture method report. Technical report, AL-TR-1995, Knowledge Based Systems, Inc., Texas, USA, 1995.

T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

National Institute of Standards and Technology, Gaithersburg, MD, USA. *Integration definition for function modeling (IDEF0)*, 1993. Draft Federal Information Processing Standards Publication 183.

G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization.* John Wiley & Sons Inc., New York, 1988.

E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42:797–813, 1996.

J. L. Peterson. *Petri net theory and the modeling of systems.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.

C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

M. Pinedo. *Scheduling: theory, algorithms, and systems.* Prentice-Hall, Englewood Cliffs, N.J., 1995.

E. Pinson. The job shop scheduling problem: A concise survey and some recent developments. In P. Chrétienne, E. G. Coffman, J. K. Lenstra, and Z. Liu, editors, *Scheduling theory and its applications*, pages 277–293. John Wiley & Sons Ltd., Chichester, 1995.

J. M. Proth and N. Sauer. Scheduling of piecewise constant product flows: A Petri net approach. *European Journal of Operational Research*, 106(1):45–56, 1998.

S. E. Ramaswamy and S. B. Joshi. Deadlock-free schedules for automated manufacturing workstations. *IEEE Transactions on Robotics and Automation*, 12(3): 391–400, 1996.

P. Richard and C. Proust. Solving scheduling problems using Petri nets and constraint logic programming. *RAIRO Recherche Opérationnelle*, 32(2):125–143, 1998.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Upper Saddle River, N.J., 1995.

H. Shih and T. Sekiguchi. A timed Petri net and beam search based on-line FMS scheduling system with routing flexibility. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 2548–2553, 1991.

S. F. Smith. Reactive scheduling systems. In D. E. Brown and W. T. Scherer, editors, *Intelligent Scheduling Systems*, pages 155–192. Kluwer Academic Publishers, Boston, 1995.

T.-H. Sun, C.-W. Cheng, and L.-C. Fu. A Petri net based approach to modeling and scheduling an FMS and a case study. *IEEE Transactions on Industrial Electronics*, 41(6):593–601, 1994.

R. J. M. Vaessens, E. H. L. Aarts, and J. K. Lenstra. Job shop scheduling by local search. *INFORMS Journal on Computing*, 8(3):302–317, 1996.

W. M. P. van der Aalst. Petri net based scheduling. *OR Spektrum*, 18(4):219–229, 1996.

M. G. A. Verhoeven. Tabu search for resource-constrained scheduling. *European Journal of Operational Research*, 106(2–3):266–276, 1998.

J. Wang, P. B. Luh, X. Zhao, and J. Wang. An optimization-based algorithm for job shop scheduling. *SADHANA*, 22(2):241–256, 1997.

G. Weiss. A tutorial in stochastic scheduling. In P. Chrétienne, E. G. Coffman, J. K. Lenstra, and Z. Liu, editors, *Scheduling theory and its applications*, pages 33–64. John Wiley & Sons Ltd., Chichester, 1995a.

M. A. Weiss. *Data Structures and Algorithm Analysis.* Benjamin/Cummings Publishing Company, Inc., California, USA, second edition, 1995b.

P. H. Winston. *Artificial Intelligence.* Addison Wesley, Massachusetts, USA, second edition, 1984.

H. H. Xiong and M. C. Zhou. Deadlock-free scheduling of an automated manufacturing system based on Petri nets. In *Proceedings of the 1997 IEEE International Conference on Robotics and Automation*, pages 945–950, 1997.

H. H. Xiong and M. C. Zhou. Scheduling of semiconductor test facility via Petri nets and hybrid heuristic search. *IEEE Transactions on Semiconductor Manufacturing*, 11(3):384–393, 1998.

H. S. Yan, N. S. Wang, X. Y. Cui, and J. G. Zhang. Modeling, scheduling and control of flexible manufacturing systems by extended high-level evaluation Petri nets. *IIE Transactions*, 29(2), 1997.

M. Zweben and M. S. Fox, editors. *Intelligent Scheduling.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.