A Genetic Algorithm for Resource-Constrained Scheduling

by

Matthew Bartschi Wall

B.S. Mechanical Engineering
Massachusetts Institute of Technology, 1989

M.S. Mechanical Engineering
Massachusetts Institute of Technology, 1991

M.S. Management
Sloan School of Management, 1991

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Mechanical Engineering
at the Massachusetts Institute of Technology

June 1996

Signature of Author: ________________________________
Department of Mechanical Engineering
14 May 1996

Certified by: ________________________________
Mark Jakiela
Associate Professor of Mechanical Engineering
Thesis Supervisor

Certified by: ________________________________
Woodie C. Flowers
Pappalardo Professor of Mechanical Engineering
Thesis Supervisor

Accepted by: ________________________________
Ain A. Sonin
Chairman, Department Committee on Graduate Students

A Genetic Algorithm for Resource-Constrained Scheduling

by Matthew Bartschi Wall

Submitted to the Department of Mechanical Engineering on 14 May 1996 in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Mechanical Engineering

Abstract

This work describes a genetic algorithm approach to resource-constrained scheduling using a direct, time-based representation. Whereas traditional solution methods are typically sequence-based, this representation encodes schedule information as a dual array of relative delay times and integer execution modes. The representation supports multiple execution modes, preemption, non-uniform resource availability/usage, a variety of resource types, probabilistic resource performance models, overlapping precedence relationships, and temporal constraints on both tasks and resources. In addition, the representation includes time-varying resource availabilities and requirements. Many objective measures can be defined such as minimization of makespan, maximization of net present value, or minimization of average tardiness. Multiple, possibly conflicting objectives are supported. The genetic algorithm adapts to dynamic factors such as changes to the project plan or disturbances in the schedule execution.

In addition to the scheduling representation, this thesis presents a structured method for defining and evaluating multiple constraints and objectives.

The genetic algorithm was applied to over 1000 small job shop and project scheduling problems (10-300 activities, 3-10 resource types). Although computationally expensive, the algorithm performed fairly well on a wide variety of problems. With little attention given to its parameters, the algorithm found solutions within 2% of published best in 60% of the project scheduling problems. Performance on the jobshop problems was less encouraging; in a set of 82 jobshop problems with makespan as the single performance measure, the algorithm found solutions with makespan 2 to 3 times the published best. On project scheduling problems with multiple execution modes, the genetic algorithm performed better than deterministic, bounded enumerative search methods for 10% of the 538 problems tested.

The test runs were performed with minimal attention to tuning of the genetic algorithm parameters. In most cases, better performance is possible simply by running the algorithm longer or by varying the selection method, population size or mutation rate. However, the results show the flexibility and robustness of a direct representation and hint at the possibilities of integrating the genetic algorithm approach with other methods.

Thesis Committee:

Mark Jakiela† , *Associate Professor of Mechanical Engineering, MIT*
Woodie Flowers, *Pappalardo Professor of Mechanical Engineering, MIT*
Stephen Graves, *Professor of Management Science, Sloan School of Management*
Karl Ulrich, *Associate Professor of Operations and Information Management, The Wharton School*

This document is available from ftp://lancet.mit.edu/pub/mbwall/phd/thesis.ps.gz

---

† effective 1 August 1996, Hunter Associate Professor of Mechanical Design and Manufacturing, Mechanical Engineering, Washington University, St. Louis.

Funding for this work was provided by the Leaders for Manufacturing Program.

1. Introduction

Planning and scheduling are common to many different engineering domains. Whether the project is as large as the Boston Harbor Tunnel or something as seemingly simple as the redesign of the packaging for a tape dispenser, both planning and scheduling are profoundly important. Even on a small project, the number of possible courses of action and the number of ways to allocate resources quickly become overwhelming. On a factory floor, determining which jobs should be executed on which machines by which employees can mean the difference between significant profit or debilitating loss. In a software development shop, assigning responsibility for tasks and effectively managing disruptions can mean the difference between a product that ships in time to hit a market window and a product that misses that window.

This document describes a genetic algorithm for finding optimal solutions to dynamic resource-constrained scheduling problems. Rather than requiring a different formulation for each scheduling problem variation, a single algorithm provides promising performance on many different instances of the general problem. Whereas traditional scheduling methods use search or scheduling rules (heuristics) specific to the project model or constraint formulation, this method uses a direct representation of schedules and a search algorithm that operates with no knowledge of the problem space. The representation enforces precedence constraints, and the objective function measures both resource constraint violations and overall performance.

In its most general form, the resource-constrained scheduling problem asks the following: Given a set of activities, a set of resources, and a measurement of performance, what is the best way to assign the resources to the activities such that the performance is maximized? The general problem encapsulates many variations such as the job-shop and flowshop problems, production scheduling, and the resource-constrained project scheduling problem.

Scheduling requires the integration of many different kinds of data. As illustrated in Figure 1, constructing a schedule requires models of processes, definition of relationships between tasks and resources, definition of objectives and performance measures, and the underlying data structures and algorithms that tie them all together. Schedules assign resources to tasks (or tasks to resources) at specific times. Tasks (activities) may be anything from machining operations to development of software modules. Resources include people, machines, and raw materials. Typical objectives include minimizing the duration of the project, maximizing the net present value of the project, or minimizing the number of products that are delivered late.

Planning and scheduling are distinctly different activities. The plan defines *what* must be done and restrictions on *how* to do it, the schedule specifies both *how* and *when* it will be done. The plan refers to the estimates of time and resource for each activity, as well as the precedence relationships between activities and other constraints. The schedule refers to the temporal assignments of tasks and activities required for actual execution of the plan. In addition, any project includes a set of objectives used to measure the performance of the schedule and/or the feasibility of the plan. The objectives determine the overall performance of the plan and schedule.

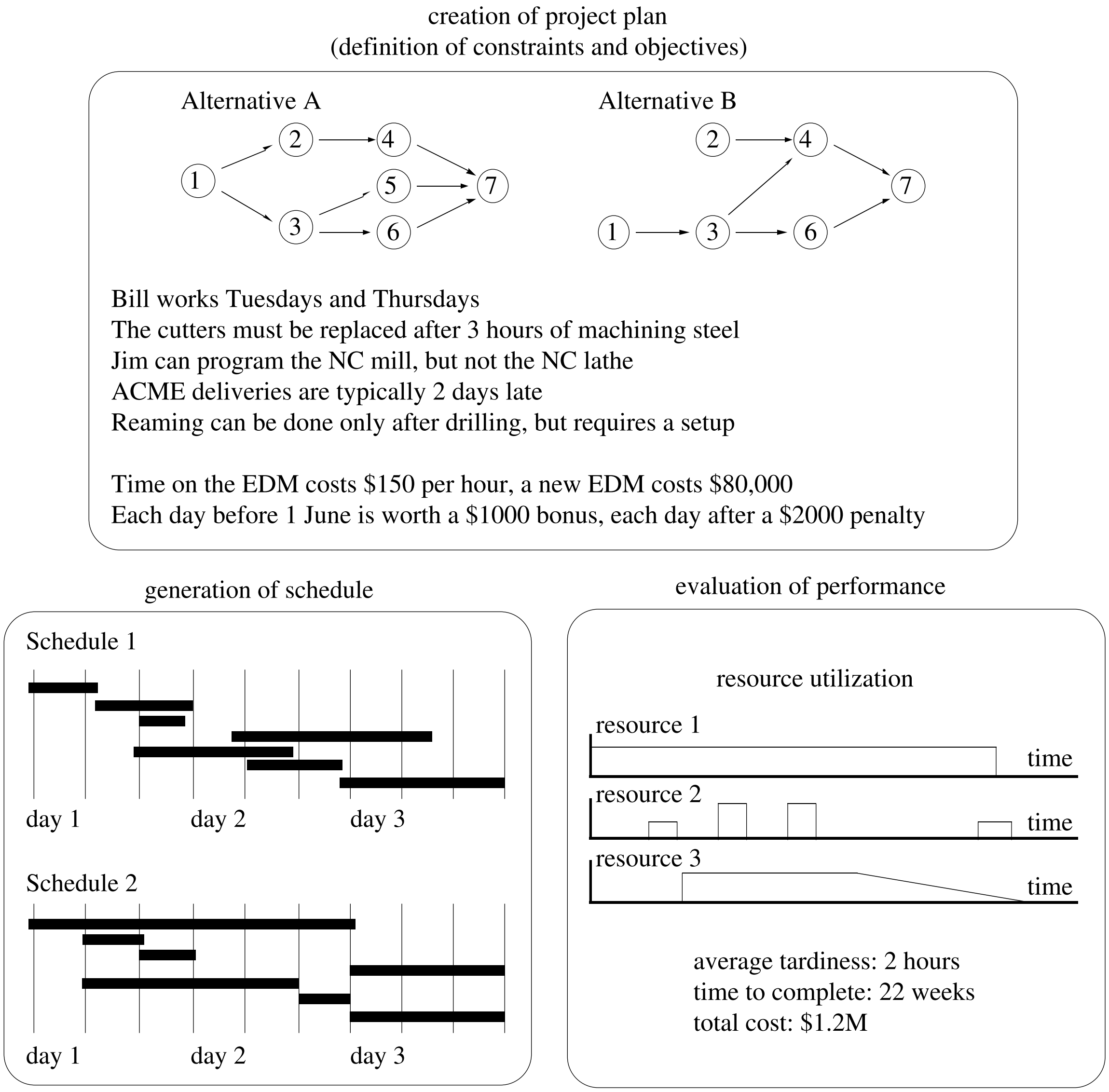


Figure 1 The parts of a resource-constrained scheduling problem. A plan is a set of constraints that define the problem. A schedule consists of a set of assignments of resources to activities at specific times. Resources include people, machines, and raw materials. Constraints define the limits of resources and relations between activities. Objectives define the goals and performance measures for the problem.

Although often treated separately, planning and scheduling are often inseparably connected. Changes to a schedule may require a different set of activities in order to produce a feasible schedule. Conversely, a plan may have no corresponding feasible schedule. In either case, objectives such as “minimize makespan” or “maximize net-present value of the cash flows”, while independent of the plan or schedule, determine the value of a plan and schedule. Both the plan and the objectives determine the difficulty of finding a schedule.

Scheduling problems are dynamic and are based on incomplete data. No schedule is static until the project is completed, and most plans change almost as soon as they are announced. Depending on the duration of the project, the same may also be true for the objectives. The dynamics may be due to poor estimates, incomplete data, or unanticipated disturbances. As a result, finding an optimal schedule is often confounded not only by meeting existing constraints but also adapting to additional constraints and changes to the problem structure.

Scheduling problems include many types of constraints. Constraints appear in many forms: *temporal constraints* such as “James can work only on Tuesdays, Thursdays, and Fridays”; *precedence constraints* such as “The design of the interface can be started when library programming interface is frozen and the analog-to-digital hardware is 75% completed”; *availability constraints* such as “three skilled machinists are available in the second shift, four are available in the third shift”; and *combinations* such as “The injection molding machine can run three shifts between maintenance cycles”. Constraints turn a relatively smooth solution space with many optima to a very non-uniform space with few feasible solutions. A typical plan includes many bottlenecks with little flexibility for change, as well as parts that are almost unconstrained.

This is not a new subject. Planning and scheduling methods have been proposed and analyzed since at least the 1950s. Although methods exist for finding optimal solutions to some specific scheduling problem formulations, many methods do not work when the structure of the constraints or objectives change. For example, a scheduling heuristic that says “schedule the task that uses the least number of resources” may not perform well when the problem is modified to include different types of resources. In addition, many methods do not perform well when faced with problems of significant size. In many cases, simply finding feasible solutions is a considerable challenge. The difficult nature of resource-constrained scheduling led Tavares and Weglarz [Tavares & Weglarz 90] to label project management and scheduling "a permanent challenge for operations research".

In general, scheduling problems are NP-hard, meaning there are no known algorithms for finding optimal solutions in polynomial time. Algorithms exist for solving exactly some forms of the problem, but they typically take too long (i.e. more than polynomial time) when the problem size grows or when additional constraints are added. As a result, most research has been devoted to either simplifying the scheduling problem to the point where some algorithms can find solutions, or to devising efficient heuristics for finding good solutions. In some cases, the problem may consist of simply finding a feasible solution, and often the existence of a feasible solution may not be assured.

Many solution methods have been proposed and implemented. Early approaches solved simplified versions of the problem exactly, but researchers quickly realized that real problems are too large and complicated for any exact solution. For example, decision trees were used to enumerate every possible choice. Heuristic methods were then devised to find good solutions, or to find simply feasible solutions for the really difficult problems. Most research now consists of designing better heuristics for specific instances of scheduling problems. However, heuristic solutions are typically limited to a specific set of constraints or problem formulation, and devising new heuristics is difficult at best.

The complex, combinatorial nature of most scheduling problems has led many researchers to experiment with genetic algorithms as a solution method. Commonly touted for their ability to solve nonlinear and combinatorial problems, genetic algorithms typically perform well on problems in which the objective and/or search space combine both discrete and continuous variables. They are also often noted for searching large, multi-modal spaces effectively since they operate on a population of solutions rather than on one individual and use no gradient or other problem-specific information.

Genetic algorithms are a stochastic search method introduced in the 1970s in the United States by John Holland [Holland 76] and in Germany by Ingo Rechenberg [Rechenberg 73]. Based on simplifications of natural evolutionary processes, genetic algorithms operate on a population of solutions rather than a single solution and employ heuristics such as selection, crossover, and mutation to evolve better solutions.

Although genetic algorithms have been studied for over 25 years, implementing them is often as much an art as designing efficient heuristics. Much of the genetic algorithm literature is devoted to relatively simple problems. Simplistic application of a genetic algorithm to small problems often produces reasonable results, but naive application of genetic algorithms to larger problems often results in poor performance. This is due to both the nature of the genetic search and the relationships between a genetic representation and the genetic operators. Direct representation of problems, i.e. use of data types other than bit strings, promises further improvements in genetic algorithm applicability, robustness, and performance. Continued reduction in computational cost along with increases in power and speed make genetic algorithms viable alternatives despite their significant computational overhead.

Due to the continued challenge of resource-constrained scheduling and the promising performance of genetic algorithms on similar problems, scheduling problems have attracted a great deal of attention in the genetic algorithm community in the past 5 years. However, most implementations are variations of traditional operations research approaches to solving scheduling problems.

In an attempt to expand the generality of genetic algorithms, this work uses a problem-specific representation with specialized crossover and mutation in concert with domain-independent genetic algorithms. The representation has been generalized to the point where it can be used with a wide variety of scheduling problems, but it has been specialized to the structure of scheduling problems in order to improve the genetic algorithm performance. The result is an algorithm that works with many different instances of the resource-constrained scheduling problem.

The second chapter of this document is a description of the resource-constrained scheduling problem and a summary of some of its variations. The third chapter contains a brief overview of traditional and not-so-traditional techniques for finding feasible and optimal schedules for various instances of the general problem. The fourth chapter is a description of the genetic algorithm and schedule-specific representation. The fifth chapter describes the jobshop and project scheduling problems on which the genetic algorithm was run and summarizes the algorithm performance. Finally, the sixth chapter offers conclusions and suggestions for future work. A glossary of terms is included in the Appendix.

2. The Resource-Constrained Scheduling Problem

Although related and often tightly coupled, planning and scheduling are distinctly different activities. Planning is the construction of the project/process model and definition of constraints/objectives. Scheduling refers to the assignment of resources to activities (or activities to resources) at specific points in, or durations of, time. The definition of the problem is thus primarily a planning issue, whereas the execution of the plan is a scheduling issue. Yet planning and scheduling are coupled; the performance of the scheduling algorithm depends on the problem formulation, and the problem formulation may benefit from information obtained during scheduling.

Section 2.1 contains a description of the general formulation of resource-constrained scheduling, Section 2.2 details specific characteristics of the problem, Section 2.3 describes two instances of the general problem, the job-shop scheduling problem and the project scheduling problem, and Section 2.4 highlights some of the factors that make scheduling a difficult problem. A glossary of terms is included in the Appendix.

2.1 The Problem (General Formulation)

In its most general form, the resource-constrained scheduling problem is defined as follows:

> Given
>
> > a set of activities that must be executed,
> >
> > a set of resources with which to perform the activities,
> >
> > a set of constraints which must be satisfied, and
> >
> > a set of objectives with which to judge a schedule's performance,
>
> what is the best way to assign the resources to the activities at specific times such that all of the constraints are satisfied and the best objective measures are produced?

The general form includes the following characteristics:

- each task may be executed in more than one manner, depending on which resource(s) is (are) assigned to it
- task precedence relationships may include overlap so that a given task may begin when its predecessor is partially complete
- each task may be interrupted according to a pre-defined set of interruption modes (specific to each task), or no interruption may be allowed
- each task may require more than one resource of various types
- a task's resource requirements may vary over the duration of the task
- the resources may be renewable (e.g. labor, machines) or non-renewable (e.g. raw materials)
- resource availability may vary over the duration of the schedule or task
- resources may have temporal restrictions

In order to accurately model the uncertainty common in real problems, the general formulation includes the following dynamic characteristics:

- resource availabilities may change
- resource requirements may change
- objectives may change

The following sections clarify these characteristics by viewing the problem from three perspectives: the tasks, the resources, and the objectives.

2.2 Characteristics of the Generalized Form

2.2.1 Tasks (Activities)

Tasks have measurable estimates of performance criteria such as duration, cost, and resource consumption. Any task may require a single resource or a set of resources, and the resource usage may vary over the duration of the task. The estimates of duration and cost may be dependent upon the resource(s) applied to (or used by) the task. The performance measures may be probabilistic or deterministic.

A task may have multiple execution modes. Any task may be executed in more than one manner depending upon which resources are used to complete it. For example, if two people are assigned to a job it may be completed in half the time required were it done by a single person. Alternatively, a part might be manufactured using one of three different processes, depending on which machine tools are available.

A single task may be composed of multiple parts. The definition of the parts includes a specification for whether or not the task can be interrupted during the parts or only between parts. For example, a milling operation may require one setup time when performed on a milling machine or a different setup time when performed on a mill-turn machine. Setup can be aborted at any time, but once machining has begun, the task cannot be interrupted until the milling process is complete.

Often the mode includes additional information that leads to further constraints. Some tasks, once begun, may not be stopped nor their mode switched until the task is complete. Alternatively, some tasks may be aborted at any time, possibly with some corresponding cost.

Interruption modes may depend on the resources that are applied to the task. Some tasks may be interrupted, but the resources they use cannot be used elsewhere until the task is finished. Other tasks may be interrupted, the resources reassigned, then any resource reapplied when the task is resumed. It may be possible to move a resource from one task to another once the first task was started. For other tasks, a resource once committed to a task must remain with that task until it is finished. Some tasks may be interrupted then restarted later, but with some cost or degradation in performance or increase in estimated time to completion.

Tasks may use resources in constant or variable amounts for the duration of the task. They may use a fixed amount as a function of time (e.g. 1 person and 1 milling machine), a variable amount (e.g. $100 for each hour the task is executed), or they may use a fixed amount for the total duration of the activity (e.g. use $500 at start of task). Tasks may be defined that use some combination of resource types and uses.

2.2.2 Resources

Resources may be renewable or non-renewable. Renewable resources are available each period without being depleted. Examples of renewable resources include labor and many types of equipment. Non-renewable resources are depleted as they are used. Examples of non-renewable resources include capital and raw materials. Note that the distinction between renewable and non-renewable resources may be tenuous. In some cases, renewable resources may become non-renewable resources, in others, non-renewable resources may be considered renewable.

Resources vary in capability, cost, and other performance measures. For example, each vendor may have an associated likelihood of on-time delivery. One work team may be more efficient than another.

Resource availability may vary. Resources may become unavailable due to unforeseen interruptions, failures, or accidents.

Resources may have additional constraints. Many resources include temporal restrictions that limit the periods of time when they can be used. For example, one team of machinists may be available only during the first shift. The constraints may be more complicated as well. Another team of machinists may be available during any shift at a higher labor rate and on the condition that they receive one shift off for every two shifts on.

2.2.3 Constraints and Objectives

Constraints and objectives are defined during the formulation of the problem. Constraints define the *feasibility* of a schedule. Objectives define the *optimality* of a schedule. Whereas objectives *should* be satisfied, constraints *must* be satisfied. Both constraints and objectives may be task-based, resource-based, related to performance measures, or some combination of these.

A *feasible* schedule satisfies all of the constraints. An *optimal* schedule not only satisfies all of the constraints, but also is at least as good as any other feasible schedule. Goodness is defined by the objective measures. When modeling the problem it is often convenient to think of objectives and constraints as equivalent, but when solving the problem they must be treated differently.

Project scheduling problems typically specify the minimization of project duration as the primary objective. However, most real problems are subject to multiple, often conflicting, objectives. Other objectives include minimization of cost, maximization of the net present value of the project, resource utilization, resource efficiency, number of due dates that were met or not met, and minimization of work-in-progress.

Often the objectives conflict. For example, it may be easy to shorten a project's duration by assigning more expensive resources to work on the tasks, but then the cost of the project increases. As more objectives are considered, the possibility for conflicts increases. Consideration of multiple objectives requires the definition of a mechanism for defining the relationship between conflicting objectives in order to make decisions about which objectives are more important.

Constraints appear in many forms. Precedence constraints define the order in which tasks can be performed. For example, the manufacture of a part may require that drilling be done only after machining a reference plane. Similarly, the design of a feeder mechanism may begin when design of the hopper is 30% complete. Temporal constraints limit the times during which resources may be used and/or tasks may be executed. For example, backups can be done only after everyone has left the office, or Bill can work only on Tuesdays.

2.2.4 Dynamic Variations

From the time they are first defined, most plans and schedules are destined to change. A project plan is static only when the project has been completed. The schedule for a machine shop is often modified within the first hour of a shift. Many job shop schedules change because of uncertainties in arrival times or due to unexpected equipment failures. Many project plans require modification when initial estimates are found to be inaccurate or when unexpected delays confound resource availabilities. Both planning and scheduling systems must be able to adapt to changes.

An important part of the stability of a plan is the consistency of the optimization given a partially completed project. Assuming that the planning tool provides an interface that lets the user maintain data integrity, an optimizer must be able to use the existing work as a constraint (or as part of the objective measure) as it determines a new optimal schedule. Another desirable characteristic is the ability to 'freeze' part of the project schedule and optimize the remainder.

2.3 Instances of the generalized problem

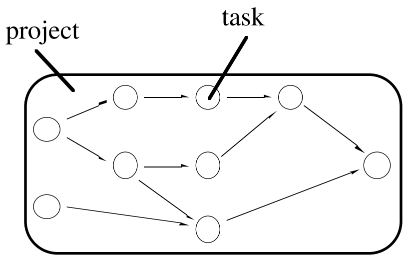
As noted by Sprecher, the flow-shop, job-shop, open-shop, and assembly line balancing problems are all instances of the general resource-constrained problem [Sprecher 94]. In addition, many production scheduling problems, single-mode resource-constrained project scheduling, multi-mode resource-constrained project scheduling, and multiple-project scheduling are also variations of the general problem.

The tests in Chapter 5 and the majority of this paper refer to two variations of the general problem, the project scheduling problem and the job-shop scheduling problem.

2.3.1 Project Scheduling

In *project scheduling problems*, a single project consists of a set of tasks, or activities. The tasks have precedence relationships, i.e. some tasks cannot be started until their predecessors have been completed. The tasks also have estimated durations and may include various other measures such as cost. Perhaps the most common objective in the project scheduling problem is to minimize the time to complete the entire project.

Many specializations of the project scheduling problem have been defined. In *resource-constrained project scheduling problems*, the tasks have resource requirements and the resources are limited. In *multi-modal resource-constrained project scheduling problems*, each task may be executed in more than one mode, and each mode may have different resource requirements. In *multi-project scheduling problems*, more than one project must be scheduled.



Precedence relations for the tasks. Tasks are represented by the circles (nodes). Precedence relations are represented by the lines (arcs).

task	task information
fabricate molds	mode 1: NC fabrication: Marc (trained to use NC mill) 1 3-axis NC milling machine 3 aluminum blanks 3 hours programming 1 hour setup 6 hours milling 4 hours finishing mode 2: non-NC fabrication: Marc, Mark, or Matthew 1 Bridgeport mill 3 aluminum blanks 4 hour setup 18 hours milling 4 hours finishing
design wing surfaces	Mark (expert with Xfoil) 4 hours programming 2 hours simulation/testing
fabricate wings	2 people required 1 hour mold preparation 1 hour materials preparation 2 hours lay-up 24 hours unattended cure

Figure 2 Parts of a project scheduling problem. A single problem may consist of multiple projects. Each project contains a set of tasks. Tasks may include many different kinds of constraints in the form of resource requirements, temporal restrictions, and precedence relations. Most tasks include measures related to performance such as quality of product, estimated duration, and cost of materials. Typical resources include people, machines, and raw materials, but may also include physical locations.

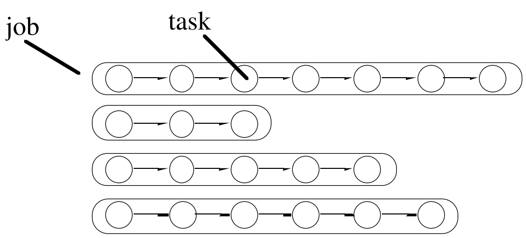
To some extent, the multi-mode project scheduling problem mixes planning with scheduling. In the simple forms of the multi-mode problems, each task may be executed in more than one mode, so this is the same as providing a set of plans and choosing between them. In a more complicated version of multi-modal project scheduling, if each task can be exploded into more tasks, or if sets of tasks can be interchanged, then the scheduler effectively does an even more complex combinatorial planning.

The domain of the project is not critical, but it does determine somewhat the complexity of the problems. It also determines the granularity of the task definition and the time scale. For example, a construction project typically has a duration measured in months, each task is often a composite of many smaller activities, and the resources are contractors and vendors. A software development project, on the other hand, has a time scale measured in weeks and the resources are individuals on the development team.

2.3.2 Job-Shop Scheduling

The typical job-shop problem is formulated as a work order that consists of set of n jobs, each of which contains m_i tasks. Each task has a single predecessor and requires a certain type of resource. Often many resources of a specific type are available, for example five milling machines and two lathes. Many tasks can be assigned to any one of the available resources, but the resource must be of the right type. Typical objectives include minimizing the makespan for the work order or meeting due dates for specific jobs or tasks.

The job-shop problems evaluated in Chapter 5 are *n* by *m*, where *n* is the number of jobs and *m* is the number of tasks per job. In these problems, *m* is also the number of (identical) machines (resources). Each task has a single execution mode, and each task requires only one resource. Each task has its own estimated duration.



job	tasks
NASA 1	rough cut block mill 5 major surfaces turn probe mount mill flat on probe mount wire EDM slot for sensor
NASA 2	rough cut NC machine features plate part
Acme Corp.	cast for rough form mill flats and ream holes anodize

Figure 3 A jobshop scheduling problem. The process plan for a single job is typically serial since each job is often associated with a single part. Each task typically requires a single resource. However, more complicated relationships are possible. The order in which jobs are executed is often unimportant in terms of the jobs themselves, but very important in terms of the resources used to do them.

Many variations are possible for either the job-shop or project scheduling problems. Some of these variations include job-splitting, task preemption, multiple execution modes, non-uniform resource availability and usage, and various resource types. These variations are described in Section 4.1

2.4 What makes scheduling problems hard?

Aside from the sheer volume of data and management of information required to schedule a project or machine shop, there are some inherent difficulties to solving even simplified scheduling problems.

2.4.1 Scaling Issues - The Size of the Problem

The size of a scheduling problem can be approximated by a what-where-when matrix. Using the nomenclature of Van Dyke Parunak, scheduling problems consist of asking *what* must be done *where* and *when*. Resources (where) operate on tasks (what) for specific periods of time (when). Using this simple classification, and neglecting precedence and other constraints, a rough approximation of a problem's size can be given by the product of what, where and when for the problem. How many tasks must be completed, by how many resources, over what time intervals?

There are many ways to prune the size of the search space. Many methods have been designed for determining whether parts of a schedule can be feasible given partial knowledge about that schedule, or whether one part of a decision tree can be any better than another part. These methods attempt to reduce the size of the search by taking advantage of problem-specific information Nevertheless, pruning heuristics are not always available, and rarely are they obvious.

The choice of representation also controls the size of the search space. If one chooses a very general representation, more types of problems may be solved at the expense of searching a larger space. Conversely, one may choose a very specific representation that significantly reduces the size of the search, but will work on only a single problem instance.

2.4.2 Uncertainty and the Dynamic Nature of Real Problems

Practically speaking, finding an optimal schedule is often less important than coping with uncertainties during planning and unpredictable disturbances during schedule execution. In some cases, plans are based upon well known processes in which resource behaviors and task requirements are all well known and can be accurately predicted. In many other cases, however, predictions are less accurate due to lack of data or predictive models. In these cases the schedule may be subject to major changes as the plan upon which it is based changes.

In either case, unanticipated disturbances to the schedule may occur. Whether a mechanical failure, human error, or inclement weather, disturbances are inevitable. Such disturbances may require only the replacement of a single resource, or they may require complete reformulation of the plan.

Any optimization technique should be able to adapt to changes in the problem formulation while maintaining the context of work already completed.

2.4.3 Infeasibility - Sparseness of the Solution Space

Depending on the representation and the modeling assumptions, there may be no feasible solution to a scheduling problem. For example, if all resources are available in constant per-period amounts and there are no temporal restrictions on tasks or resources, a project is guaranteed to have a feasible schedule. At the very worst, one need only extend the project until all tasks are completed. If, on the other hand, resources, once used, are gone forever and tasks may be executed only at certain times or within certain time limits, a feasible solution is not guaranteed. Some algorithms are capable of determining if such an infeasible situation exists. Most heuristic methods cannot.

Constraints make the search for an optimal solution more difficult by breaking up an otherwise continuous search space. When many constraints are added, traversal of the search space is confounded. In addition, adding constraints typically reduces the number of feasible solutions for a given representation.

3. Related Work

Variations of the resource-constrained scheduling problem have been proposed, implemented, and evaluated for over fifty years. The solution methods form two distinct classes: exact methods and heuristic methods. These classes may be categorized further into stochastic and deterministic approaches. Exact methods are guaranteed to find a solution if it exists, and typically provide some indication if no solution can be found. Heuristic solutions may have no such guarantee, but typically assure analytically some degree of optimality in their solutions. Stochastic methods include probabilistic operations so that they may never operate the same way twice on a given problem (but two different runs may result in the same solution). Deterministic methods operate the same way each time for a given problem. Many hybrid methods exist that combine the characteristics of these classes.

When resource-constrained scheduling solutions were first proposed, simple models were used with exact methods for solving the problem. Given a problem, the exact methods find the best solution (and are guaranteed to find the best solution) every time they are run. However, as constraints were added, the difficulty of solving the problem increased, and simply finding a good solution (or in some cases, a feasible solution) became good enough. In addition, many methods take too long when applied to problems of significant size. For example, the critical path method (CPM) was devised for finding the shortest time to complete a project given estimates of task durations. The CPM can highlight the activities in a project that will have the most effect on the completion time of the project should they take longer than predicted to execute. However, the CPM cannot solve problems that include restrictions on the number of resources that are available.

Many methods focused exclusively on scheduling and assumed that the plan was static. Stochastic models were added to account for the uncertainty of real schedules. Later methods attempted to integrate planning and scheduling in an effort to find optimization possibilities that take advantage of the inherent coupling between planning and scheduling.

Three general procedures for solving resource-constrained scheduling problems are shown in Figure 4. Specific variations of these methods are clarified in Sections 3.2 and 3.3. Some solution methods generate a sequence of tasks then schedule the tasks based on that sequence (Figure 4a). Others schedule the tasks directly by focusing on other constraints such as resource availability (Figure 4b). Some methods mix planning and scheduling by allowing the scheduler to choose from a set of process plans (Figure 4c). Some methods focus on resource constraints, others focus on precedence or temporal constraints. Some are deterministic; they find the same solution each time they run. Others are stochastic; they may find a different solution each time they are run. Many methods are hybrids that combine characteristics of these techniques.

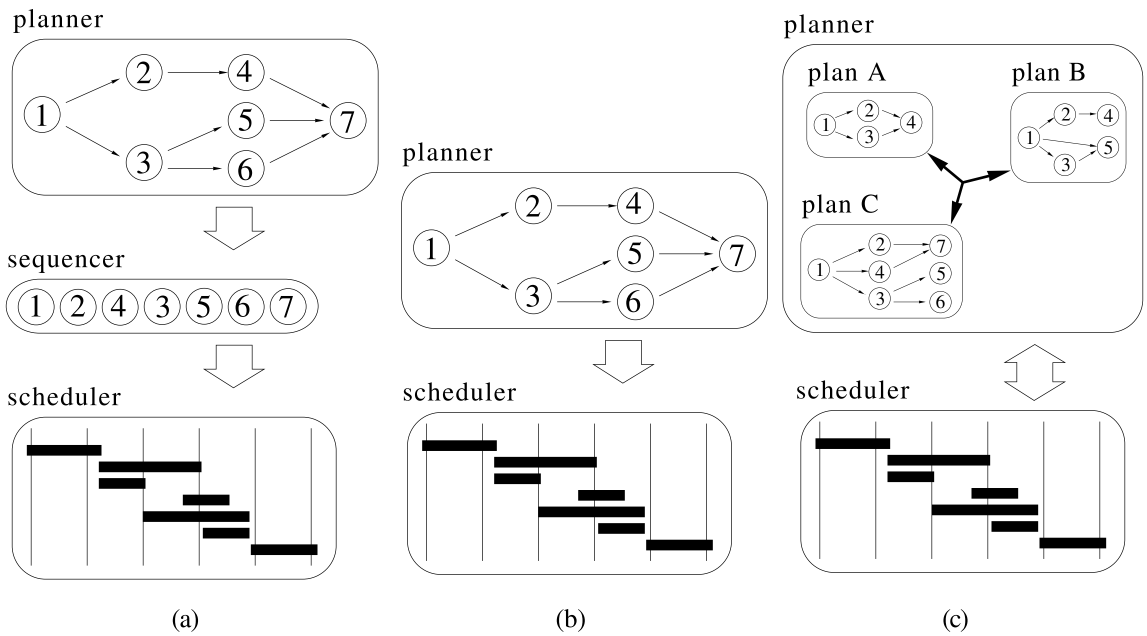


Figure 4 General procedures for three classes of scheduling algorithms. Some algorithms generate a precedence-feasible sequence of tasks then schedule the sequence (a), others schedule tasks directly from the plan by considering both precedence and resource constraints while they schedule (b), and others combine both planning and scheduling by modifying the plan to adapt to the schedule as well as the schedule to adapt to the plan (c). Planning is the definition of constraints and is represented here as precedence relations between tasks. Scheduling is the assignment of resources to tasks at specific times and is represented here by Gantt charts, a graphic display of task durations.

In addition to solution methods, recent work has focused on characterization of the general problem and creation of standard suites of problems on which to test algorithms. Even specialized instances of the resource-constrained scheduling problem are very complicated, so simply formulating the problem is non-trivial. As a result, most of the published problems are specialized, simplified, instances of the general problem. Section 3.3 highlights some of these efforts.

3.1 Characterization of the Problems and Problem Generation

After fifty years of research directed at solving scheduling problems, many recent works have been devoted to characterizing the problems. The purpose of these explorations has been to understand the structure of scheduling problems so that problems can be generated in order to rigorously test the many solution methods. Definition of the problem and construction of problem generators are tightly entwined; most problem generators include a variety of parameters with which to define the problem characteristics, but those parameters are often specific to a certain representation or class of problems.

One aspect of the problem faced by those who attempt to characterize scheduling problems is the sheer number of variations. Scheduling problems come in many different shapes and sizes. Often a small change results in a completely new formulation.

First introduced in 1950s during the development of the Polaris missile system, the program evaluation and review technique (PERT) is the forerunner of formal project scheduling representations [NASA 1962]. PERT is a method of characterizing precedence relationships between tasks and estimates of task requirements. PERT is not a scheduling method *per se*, but rather a method for defining some constraints and organizing information. Many solution methods depend on a PERT representation (or some derivative thereof) for their precedence constraints.

Sprecher presented a formal formulation of the single- and multi-mode project scheduling problem and noted the relation between project scheduling problems, the job-shop problem, the open shop problem, and assembly line balancing [Sprecher 94]. Sprecher included general temporal constraints and resource requirements that vary in time as well as a general formulation of many different performance measures.

Kolisch, Sprecher, and Drexl have characterized many variations of the resource-constrained project scheduling problem [Kolisch, Sprecher, Drexl 92]. They defined a set of parameters such as "resource strength" and "network complexity" that characterize the resource-constrainedness and number of precedence relationships in a project plan. *ProGen* is the problem generator they created that uses these parameters to specify the characteristics (and often difficulty) of the problems. In the problem sets described in their paper, Kolisch *et al* varied three parameters: *complexity*, based on the connectivity of the precedence relationship network; *resource factor*, a measure of the number of resource types; and *resource strength*, a measure of resource availability. Then they applied their solution method to the problems in order to correlate problem difficulty (based on their algorithm's performance) with the parameters.

In a paper devoted to the description of a combined heuristic for project scheduling, Boctor mentioned another problem generator [Boctor 94]. Boctor's set of 240 problems contains 120 50-activity problems and 120 100-activity problems. Each activity had an average of 2 predecessors, and the number of resource types varied from 1 to 4.

Many job shop, flow shop, and production scheduling problems are available and have been described in the literature [Adams et al, 1988] [Fisher and Thompson, 1963] [Lawrence, 1984] [Applegate and Cook 1991] [Storer et al, 1992] [Yamada and Nakano, 1992]. The classic jobshop problems include the 6x6 and 10x10 problems first presented by Muth and Thompson [Muth & Thompson, 1963]. Since then, many others have suggested and solved a variety of more complex variations.

A set of problems was posted on the world-wide web by Barry Fox and Mark Ringer in early 1995. The set consists of a single life-size (575 tasks, 3 types of labor resources and 14 location-based resources) problem in twelve parts. Each part encompasses a different variation of the basic formulation. The variations include changes in resource availability, job-splitting, a wide variety of temporal and location constraints, and multiple objectives. The set does not include multi-modal activities. Fox and Ringer noted the dearth of life-like problems that are publicly available [Fox & Ringer 1995]

Although many solution methods have been published, the complexity of even formulating the problem has been an obstacle to a widespread, common base of test problems other than the simplified job-shop formulations or very simple project scheduling problems. In addition, many solution methods have been tested on problems from industrial sponsors who are understandably anxious about publication of details about their operations.

3.2 Exact Solution Methods

Exact methods are guaranteed to find the optimal solution, but typically become impractical when faced with problems of any significant size or large sets of constraints.

3.2.1 Critical Path Method

The critical path method (CPM) provides the resource-unconstrained schedule for a set of precedence-constrained activities with deterministic durations. It gives the shortest possible makespan assuming infinite resources. Although useful for obtaining a rough idea of the difficulty of executing a plan, the critical path method does not consider temporal or resource constraints. Stochastic variations [Neumann 90][Slowinski and Weglarz 89] and dynamic variations [Blazewicz 83] have also been constructed in an attempt to bring the critical path method modeling assumptions closer to reality. These methods include probabilistic estimates of task duration.

3.2.2 Linear and Integer Programming

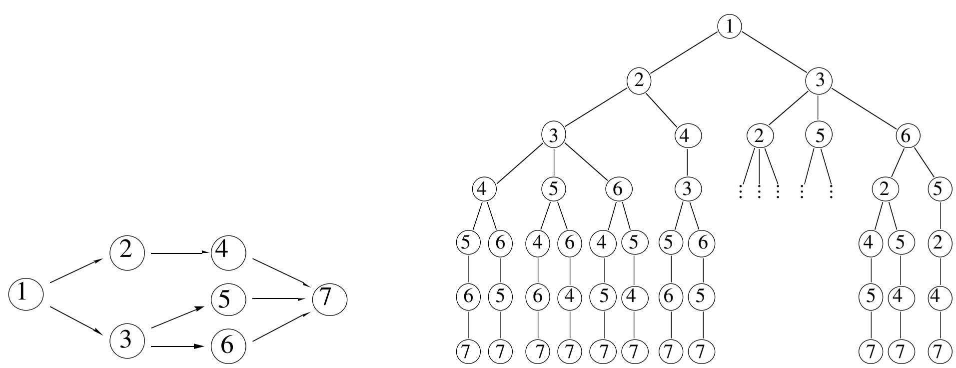
Many scheduling problems can be formulated in traditional linear or integer programming form, but only if significant simplifications are made. Patterson presented an overview of optimal solution methods for project scheduling [Patterson 84], and Demeulemeester and Herroelen published a more recent survey [Demeulemeester & Herroelen 92].

In general, exact methods depend on characteristics of the objective function (e.g. strictly integer values) and specific constraint formulations (e.g. only single-mode tasks). As Lawrence Davis noted, many of the constraints commonly found in real scheduling problems do not lend themselves well to traditional operations research or math programming techniques [Davis 85]. In addition, the linear programming formulations typically do not scale well, so they can be used only for specific instances or small problems.

A dynamic programming approach was described by Held and Karp in which an optimal schedule was incrementally developed by first constructing and optimal schedule for any two tasks then extending that schedule by adding tasks until all tasks have been scheduled [Held and Karp 62].

3.2.3 Bounded Enumeration

Many solution methods search a decision tree generated from the precedence relations in the project plan. As shown in Figure 5, the root of the tree corresponds to the first task. The second level of the tree is the set of tasks that can be scheduled once the first task has been scheduled, and so on. The final tree thus represents a precedence-feasible set of task sequences. Any one of the root-to-leaf sequences can then be passed to a schedule generator. Alternatively, the sequence of tasks can be scheduled directly if the tree generation/pruning algorithm also considers resource constraints. The search consists of traversing the tree until the best root-to-leaf path is found. Enumerative methods are typically bounded using heuristics in order to reduce the size of the tree.



(a) (b)

Figure 5 Generation of a tree of precedence-feasible sequences from a project plan (or work order). The precedence constraints are shown in (a) for a project (or work order) with 7 tasks. The tree of precedence-feasible sequences for scheduling the 7 tasks is shown in (b).

It is easy to see how the tree grows quickly with the number of activities. Depending on the precedence relations, each new task can add many branches to the tree. When tasks are modeled with multiple execution modes, each execution mode adds another layer of combinatorial choices for the scheduler. Sprecher and Drexl note that the enumerative methods can solve the problem with many different objectives [Sprecher and Drexl 1996]. However, changing the types of constraints requires a new set of heuristics for the scheduling step, or a new pruning algorithm for trimming the tree branches.

Variations of branch and bound solution methods were first proposed in the 1960s [Lawler and Wood 66][Johnson 67][Müller-Merbach 67]. Stinson's branch and bound approach generated a tree by scheduling activities starting with the first task then adding a node to the tree for each task that could be scheduled based upon precedence and resource constraints [Stinson et al 78]. Bounds based on partial schedules were used to prune the search tree. The heuristic for expanding the tree used a vector of six measures.

More recently, Sprecher, Kolisch, Drexl, Patterson, Demeulemeester, and Herroelen have refined the pruning algorithms so that optimal solutions to single-mode project scheduling problems of about 100 tasks and multi-mode project scheduling problems of about 10 tasks can be found in less than a few seconds on personal computers [Kolisch 95][Sprecher & Drexl 96]

As noted by Sprecher and Drexl, enumerative methods cannot solve large problems; the tree is simply too big Although significant progress has been made in the pruning techniques, branch and bound methods are still limited to less than one hundred activities or even fewer in the multi-modal cases, and they still require special heuristics to accommodate variations in resource constraint formulations.

3.3 Heuristic Solution Methods

Whereas exact solution methods are guaranteed to find the optimal solution (if one exists), heuristic methods sometimes find optimal solutions, but more often find simply "good" solutions. Heuristic methods typically require far less time and/or space than exact methods. The heuristics specify how to make a decision given a particular situation; heuristics are rules for deciding which action to take.

Heuristics in scheduling are often referred to as scheduling rules or dispatch rules. The definition of these rules is often quite complex, and most are tailored for a specific type of problem with a very specific set of constraints and assumptions. Heuristics may be deterministic - they end up with the same result every time - or they may be stochastic - each time they are run they may produce a different result. They may execute one rule at a time, or they may be capable of parallel decisions. Hybrid algorithms may combine multiple heuristics.

Traditional heuristic methods typically follow three steps: planning, sequencing, then scheduling. Some methods use heuristics to search the combinatorial space of permutations in task sequences, others use heuristics to determine feasible time/task/resource assignments during schedule generation, and others use heuristics to combine sequencing and scheduling. A few include planning in the generation of schedules by permitting more than one plan and allowing the search to choose between plans as it schedules. Precedence constraints typically dominate the search in the sequencer, whereas resource constraints dominate in the scheduler. Hybrid solutions try to maintain more than one representation or combine multiple search techniques or constraint satisfaction algorithms.

3.3.1 Scheduling Heuristics

Scheduling heuristics operate on a set of tasks and determine when each task should be executed. If a task may be executed in more than one execution mode or on any one of a set of resources, the heuristic must also determine which resources and/or execution mode to use. Common heuristics are listed in Table 1. The scheduler enforces constraint satisfaction by assigning a task to a resource (or a resource to a task) at a time when the resource is available and the task can be executed.

heuristic	what it does
MIN SLK	choose the task with the smallest total slack
MIN LFT	choose the task with the nearest latest finish time
SFM	choose the execution mode with the shortest feasible duration
LRP	choose the execution mode with the least resource proportion

Table 1 Some commonly-used scheduling heuristics. Dispatch rules (a form of scheduling heuristic) decide which resources should receive tasks as they come in to a shop.

Panwalker and Iskander surveyed a range of heuristics from simple priority rules to very complex dispatch rules [Panwalker and Iskander 77]. Davis and Patterson compared eight standard heuristics on a set of single-mode resource-constrained project scheduling problems [Davis and Patterson 75]. They compared the performance of the heuristics with optimal solutions found by a bounded enumeration method by Davis and Heidorn [Davis and Heidorn 1971]. The results showed that the MIN SLK heuristic performed best. The results also showed that the heuristics did not perform as well when the resources were tightly constrained.

Lawrence and Morton described an approach that attempted to minimize weighted tardiness by using a combination of project-, activity-, and resource-related metrics [Lawrence and Morton 93]. The results of their approach were compared to a large set of problems with hundreds of tasks distributed between five projects with various tardiness penalties, activity durations, and resource requirements. Their heuristic produced schedules with lower average tardiness costs than did the standard heuristics.

In his review of heuristic techniques, Hildum made the distinction between single- and multiple-heuristic approaches [Hildum 94]. While emphasizing the importance of maintaining multiple scheduling perspectives, Hildum noted that a scheduler with multiple heuristics "is better able to react to the developing multi-dimensional topology of the search space." Boctor's experiments with multiple heuristics clearly showed the benefits of combining the best of the single-heuristic methods [Boctor 90].

3.3.2 Sequencing Heuristics

Whereas scheduling heuristics operate on tasks to decide when they should be executed, sequencing heuristics determine the order in which the tasks will be scheduled. These heuristics are often used in combination with decision trees to determine which part of the tree to search or to avoid. For example, limited discrepancy search with backtracking has been used by William Harvey and Matthew Ginsberg to very effectively solve some classes of scheduling problems when the sequence for scheduling tasks is structured as a decision tree [Harvey and Ginsberg 94].

Sampson and Weiss designed a local search procedure for solving the single mode project scheduling problem [Sampson & Weiss 93]. They described a problem-specific representation, a neighborhood structure, and method for searching the neighborhood. Their deterministic method performed fairly well on the 110 Patterson problems [Patterson 84].

3.3.3 Hierarchical Approaches

Goal programming has been used to solve scheduling problems with multiple objectives. Norbis and Smith described a method for finding near-optimal schedules using levels of consecutive orderings of sub-problems [Norbis & Smith 88]. As a collection of orders moved up through the levels, the tasks were rearranged to accommodate the priority of objectives at each level. In addition to dynamic changes such as surprise orders and changes in resource availability, Norbis and Smith's implementation also allowed user input during the solution process.

3.3.4 Artificial Intelligence Approaches

Hildum grouped artificial intelligence approaches to scheduling as either expert systems or knowledge-based [Hildum 94]. Both are structured heuristic methods that differ in the way they control the application of their application-specific heuristics.

Expert systems consist of a rule base, a snapshot of the current solution, and an inference engine. The inference engine determines how the if-then rules in the rule base are applied to the current solution in order to execute the search. The rule base may be expanded as the solution progresses. The rule base is tailored explicitly to a specific problem, so expert systems are typically highly specialized.

Knowledge-based systems typically split the problem into sub-problems or different views. "Agents" are defined, each of which is concerned with a particular aspect of the solution. Each agent nudges the solution in the direction of most concern to that agent. Variations to the algorithms include combinations of micro and macro modifications to solutions as well as the types of attributes to which the agents are configured to respond. Hildum distinguished between three commonly known artificial intelligence solutions, ISIS (Intelligent Scheduling and Information System) [Fox and Smith 84], OPIS (Opportunistic Intelligent Scheduler) [Smith and Ow 85], and MicroBOSS (Micro-Bottleneck Scheduling System) [Sadeh 91], based upon the rules they used to guide their searches. Hildum noted that his own method, DSS (Dynamic Scheduling System), is basically a multiple attribute, dynamic heuristic approach that focuses on the most urgent unsolved problem at any given time.

3.3.5 Simulated Annealing

Simulated annealing approaches require a schedule representation as well as a neighborhood operator for moving from the current solution to a candidate solution. Annealing methods allow jumps to worse solutions and thus often avoid local sub-optimal solutions [Kirkpatrick et al 83]. Aarts, Laarhoven, and Lenstra described one of the first simulated annealing approaches to scheduling problems [Aarts et al 88].

Palmer integrated planning and scheduling in a digraph representation [Palmer 94]. In Palmer's representation, a graph of precedence constraints and process plans combined to form a schedule. The annealer used three operators to modify the ordering of operation-machine combinations.

Boctor reported fairly good performance by a simulated annealing approach on the Patterson problems [Boctor 93]. In this work, simulated annealing was used to search the combinatorial space of sequence permutations. Given a sequence of tasks generated by the annealer, heuristics were then used to create a schedule. This method is directly analogous to the exact branch and bound solution, but whereas branch and bound is practically limited by the size of the decision tree, simulated annealing can be applied to much larger problems. However, selection of the neighborhood operator and cooling schedule is critical to the performance of the annealing method. Boctor's implementation maintained precedence feasibility by restricting the neighborhood operator to only precedence-feasible task swaps. Precedence-feasible lists generated by the annealer were passed to a heuristic scheduler in order to generate resource-feasible schedules.

3.3.6 Evolutionary Algorithms

One of the earliest suggested uses of genetic algorithms for scheduling was made by Lawrence Davis. In his paper [Davis 85], Davis noted the attractiveness of using a stochastic search method due to the size of the search space and suggested an indirect representation in which the genetic algorithm operated on a list which was then decoded to form the actual schedule. In particular he noted the importance of maintaining feasibility in the representation.

Davis observed that many real scheduling problems incorporate layers of ill-defined constraints that are often difficult, if not impossible, to represent using traditional math programming techniques. Noting that knowledge-based solutions are typically deterministic and thus susceptible to entrapment in sub-optimal regions of the search space, Davis suggested that genetic algorithms, by virtue of their stochastic nature, would avoid sub-optimal solutions.

Since Davis' paper, numerous implementations have been suggested not only for the jobshop problem but also other variations of the general resource-constrained scheduling problem. In some cases, a representation for one class of problems can be applied to others as well. But in most cases, modification of the constraint definitions requires a different representation.

Ralf Bruns summarized the production scheduling approaches in four overlapping categories: direct, indirect, domain-independent, and problem-specific representations [Bruns 93]. Most genetic algorithm approaches employed an indirect representation. These methods were characterized by traditional binary representations [Nakano 91][Cleveland 89] or order-based representations [Syswerda 91]. Problem-specific information was used in some indirect methods to improve performance, but these were still list- or order-based, required transformation from genome to schedule, and in some cases required a schedule builder as well [Bagchi 91]. A direct representation by Kanet used a list of order-machine-time triplets, but, as noted by Bruns, was not extensible [Kanet 91]. Noting the inverse relation between the generality of an algorithm and its performance, Bruns' representation was tuned "to perform as efficiently as possible on the problem of production scheduling". This direct, problem-specific representation used a list of order assignments in which the sequence of orders was not important.

Bagchi compared three different representations and concluded that the more problem-specific information was included in the representation, the better the algorithm would perform [Bagchi 91].

More recently, Philip Husbands outlined the state-of-the-art in genetic algorithms for scheduling [Husbands 96]. Husbands noted the similarity between scheduling and sequence-based problems such as the traveling salesman problem. He also referenced other NP-hard problems such as layout and bin-packing problems that are similar to the jobshop formulation.

Of particular note is Cleveland and Smith's work in which three models were compared: a pure sequencing version, a model with release times, and a model with work-in-progress costs [Cleveland & Smith 89]. When the more-realistic objective function of the third model was used, the pure sequencing model performed poorly whereas the model with schedule information found significantly better solutions.

In a set of preliminary tests with various representations, the author tried a sequence-then-schedule approach for project scheduling similar to the simulated annealing approach of Boctor but with a genetic algorithm. The representation was a sequence-based hybrid; the genetic algorithm generated sequences then the sequences were heuristically scheduled. Results were not encouraging; mutation only (i.e. a random search) performed better than the genetic algorithm. This performance is analogous to the difference in performance between threshold accepting and a genetic algorithm for the traveling salesman problem. Boctor's simulated annealing approach performed better than the genetic algorithm primarily due to better scheduling heuristics (Boctor implemented a parallel, look-ahead scheduler). Another factor was the reordering method. In the genetic algorithm implementation, the author used partial match crossover, whereas Boctor did a random, precedence-based shuffle. As the difference between edge recombination and partial match on the traveling salesman problem illustrates, choice of crossover operator can mean the difference between a genetic algorithm that works and one that does not. Better performance is possible with the genetic algorithm, but only with a crossover operator tailored to the problem.

A number of hybrid solutions have been implemented. Syswerda and Palmucci combined a genetic algorithm sequencer with a deterministic scheduler with special order-based operators [Syswerda 90].

Genetic algorithms have been used to evolve heuristics for scheduling. Hilliard implemented a classifier system to improve heuristics for determining the sequence of activities that should be sent to a scheduler (analogous to deciding which path to choose in the decision tree) [Hilliard 88]. Hilliard's system discovered the "sort the jobs by increasing duration" heuristic for the simple one-operation-per-job, no ordering constrains, single machine shop scheduling problem.

Dorndorf and Pesch used a genetic algorithm to find optimal sequences of local decision rules to be used with traditional search algorithms for a range of static, deterministic jobshop scheduling problems [Dorndorf & Pesch 92]. Their method found shorter makespans more quickly than the shifting bottleneck procedure of Adams, Balas and Zawack [Balas 88] or Lenstra's simulated annealing method [Aarts 88].

Husbands addressed the issue of coupling between scheduling and planning by implementing an integrated system in which process plans evolved then were combined through a scheduler to build schedules that could then be evaluated [Husbands 91]. This work was important for its integration of planning and scheduling; in many cases a change in schedule will necessitate a change in the plan or result in the possibility for optimization by modifying constraints, so an integrated, adaptive system is well-suited for real scheduling applications.

Various papers have been written describing parallel genetic algorithm implementations, but most of these are straightforward extensions of serial genetic algorithms and offer little improvement in algorithmic performance. Parallelization by distributing computation will speed up execution, but additional evolutionary operations such as migration are required for improvements in solution quality.

The applications of genetic algorithms to scheduling problems are summarized in Table 2. The diagrams in the first column illustrate the basic representation used in each case. Note that these representations were not all designed to solve the same problem. Many of these representations require representation-specific operators.

representation	characteristics	reference
(order 6) (order 1) (order 4) ...	list of orders to be scheduled in a job-shop	[Syswerda 91]
1 0 1 1 0 0 1 1 1 1 0 0 ...	binary representation in which each bit determines which order of a pair should be executed first on a given machine	[Nakano 91] [Cleveland 89]
(order 2 plan X) (order 1 plan Q) (order 4 plan B) ...	list of (order, plan) pairs	[Bagchi 91]
(order 5 plan A op1 : m3 op2 : m5 op3 : m3) (order 2 plan X op1 : m1 op2 : m7) (order 1 plan A op1 : m4 op2 : m6 op3 : m7) ...	list of (order, plan, resource) tuples	[Bagchi 91]
(station 1 0 : o1 wait o4 idle 60 o2 wait o1 wait o4 ...) (station 2 0 : o4 wait o4 idle 60 o5 wait o6 idle ...) ...	list of order/time preferences for each workstation	[Davis 85]
(order 5 machine 3 00:00) (order 2 machine 6 01:05) (order 1 machine 2 00:15) ...	list of order/machine/time tuples	[Kanet 91]
(order 7 Op7B1 : m9 : [10,15] Op7B2 : m3 : [16,17] Op7B3 : m6 : [18,22]) (order 9 Op9A1 : m9 : [10,15] Op9A2 : m3 : [16,17]) ...	list of complete order information	[Bruns 93]
(task 2, mode 5 order 5 01:00 - 04:30) (task 5, mode 1 order 1 00:30 - 00:50) ...	list of (activity-mode, order, start-finish-time) tuples	[Mori and Tseng 96]
(project 1 activity 2 mode 2) (project 1 activity 3 mode 4) (project 2 activity 1 mode 1) ...	list of (project, activity, mode) tuples	[Tseng and Mori 96]

Table 2 Summary of various genetic algorithm formulations. The references in the table are representative of the type of solution; this table does not contain an exhaustive list of published works. Most of the representations are order-based, i.e. the order in which the items appear in the list is a part of the problem structure.

Although some of these methods can be extended to do so, only the representation of Tseng and Mori considers multiple execution modes. None consider non-uniform resource availabilities. Most require significant modification to accommodate job-splitting.

None of the evolutionary algorithms can determine if a schedule is infeasible. The problems for which these methods were designed all had feasible solutions, but in real problems feasibility is not guaranteed. For example, in a project scheduling problem, if all of the resources are available in constant amounts, a feasible schedule can always be found by simply extending the duration of the project until all of the resources are unconstrained. When the resources are available only at specific intervals, no such guarantee of feasibility exists.

Although Husbands compared the scheduling problem to other sequence-based methods, his analogy is not entirely appropriate. The scheduling problem is only a sequencing problem when viewed from the traditional operations research approach, i.e. what should be scheduled next? While this perspective is entirely appropriate for some types of scheduling problems, there remain many other perspectives, most of which have not yet been exploited with genetic algorithms.

Problem-specific operators often improve the performance of genetic algorithms, but so does definition of a problem-specific representation. The key is to define a representation that is general enough to accommodate all of the problem instances one wishes to solve, yet specific enough to actually work.

4. Solution Method

This chapter describes the solution method in five parts: (1) the problem model in the form of assumptions about tasks and resources with their associated constraints, (2) the search method, (3) the schedule representation, (4) the genetic operators specific to the representation, and (5) the implementation of objectives and constraints. The problem model determines the variations of problems that can be solved, the problem representation determines the bounds of the search space, the genetic operators determine how the space can be traversed, and the objectives and constraints determine the shape of the search space.

4.1 Problem Model

The assumptions made when modeling a problem determine the variations of that problem that the model will support. The next three sections list assumptions about tasks, resources, and objectives. The assumptions in the first two sections typically end up being constraints. The third section highlights some of the more common objectives that may be defined. Satisfaction of the constraints determines the *feasibility* of a solution, satisfaction of the objectives determines the *optimality* of a solution.

4.1.1 Assumptions About Tasks

assumption	description	examples
execution modes	Tasks may have more than one mode of execution. Each execution mode has its own set of resource requirements and estimated duration.	The machine will require 8 hours if done on the new milling machine or it will require 12 hours on the older milling machine. The vendor in Kentucky promises a delivery date of the fifteenth of the month, whereas the vendor in Louisiana promises delivery by the first of the month.
precedence and overlap	Tasks have precedence relationships. Normally this means that one task cannot be started until another task has been completed. However, overlap is possible; relationships may be defined in which a task may be started when its predecessor is only partially complete.	Work on the implementation of a class library may begin when the interface design is 50% complete, whereas testing of the PC version cannot begin until the UNIX version is complete.
job-splitting	Tasks may be split into smaller units. The definition of execution modes may also include modes for task splitting.	Using one process plan it may be possible to remove a fixtured part from the machine then return it later, whereas a different process may require that the part be completed once the processing has begun.
preemption	Tasks may be preempted. A task may be defined so that its resources may be applied to a different task, then returned to the original task.	Bill can stop work on the enclosure design after completing the layout in order to work on the electronics packaging, but if someone else takes over the task at that point they will have to learn the FCC regulations, so the task will take an extra two hours to complete.

assumption	description	examples
temporal restrictions	Tasks may have temporal constraints such that they can only be done at a certain time.	Work on a section of highway must be done only at night from 9 p.m. to 6 a.m.
location restrictions	Tasks may have physical restrictions.	Painting can only be done in the paint booth. Any part to be anodized must be moved to the anodization tank, whereas parts that are to be refurbished may be worked on in place.

4.1.2 Assumptions About Resources

assumption	description	examples
resource type	Resources may be renewable (non-consumable) or non-renewable (consumable).. The distinction between renewable and non-renewable is somewhat tenuous; in some cases, non-renewable resources may become renewable resources.	Labor and machines are examples of renewable resources. Non-renewable resources include raw materials. A truckload of lumber can be considered a non-renewable resource, but if another truckload can be delivered within a critical time period, the lumber may be considered renewable.
temporal restrictions	Resources may have temporal constraints that limit their use to a certain time or time period.	Machinists may be available only during the first two shifts, or a particular tractor may require two hours of maintenance for every twenty hours of use.
performance history	Resources may have performance histories from which to base estimates of performance.	One vendor might deliver 90% of its orders on time whereas another delivers 98% of its orders on time. Or one employee may be capable of building product models at one rate and web site design at a different rate.
location restrictions	Resources may be restricted to certain locations, or they may be mobile.	A telecommunications equipment truck has a range of 300 miles with a top speed of 55 mph.
task-resource dependencies	Resources may include additional availability constraints depending on the type of work they are assigned to.	A milling machine may require a one hour setup and a half hour cleanup when working with aluminum, but a four hour setup and two hour cleanup when working with Beryllium.
performance attributes	Resources have an associated cost, quality, or efficiency that can be used to evaluate performance.	Skilled labor is more expensive than unskilled labor, and time on one machine tool may be more expensive than time on another.

4.1.3 Assumptions About Objectives

Any objective measure can be used as long as it can be determined from a complete schedule. Examples of objectives include minimization of makespan, minimization of mean tardiness of part delivery times, maximization of net present value, and minimization of work-in-progress. For detailed descriptions of the implementation of some common objectives, see Section 4.5.1.3.

Objectives may include more than one part. For example, minimization of the makespan may be the primary objective, but only if it does not drive the cost above a certain threshold.

4.2 Search Method

Genetic algorithms are a stochastic heuristic search method whose mechanisms are based upon simplifications of evolutionary processes observed in Nature. Since they operate on more than one solution at once, genetic algorithms are typically good at both the *exploration* and *exploitation* of the search space. Goldberg [Goldberg 89] provided a comprehensive description of the basic principles at work in genetic algorithms, and Michalewicz [Michalewicz 94] described many of the implementational details for using genetic algorithms with various data types.

Most genetic algorithms operate on a population of solutions rather than a single solution. The genetic search begins by initializing a population of individuals. Individual solutions, or *genomes*, are selected from the population, then *mate* to form new solutions. The mating process, typically implemented by combining, or *crossing over*, genetic material from two parents to form the genetic material for one or two new solutions, confers the data from one generation of solutions to the next. Random mutation is applied periodically to promote diversity. If the new solutions are better than those in the population, the individuals in the population are replaced by the new solutions. This process is illustrated in Figure 6.

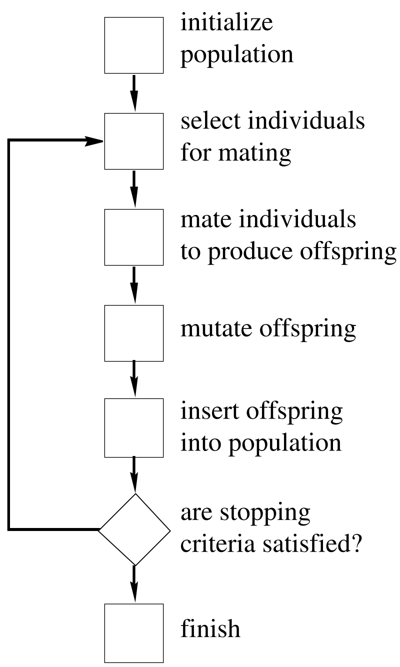


Figure 6 Generic genetic algorithm flowchart. Many variations are possible, from various selection algorithms to a wide variety of representation-specific mating methods. Note that there is no obvious criterion for terminating the algorithm. Number-of-generations or goodness-of-solution are typically used.

Genetic algorithms operate independently of the problems to which they are applied. The genetic operators are heuristics, but rather than operating in the space defined by the problem itself (the *solution-space*, or *phenotype-space*), genetic operators typically operate in the space defined by the actual representation of a solution (*representation-space*, or *gene-space*). In addition, genetic algorithms include other heuristics for determining which individuals will mate (*selection*), which will survive to the next generation (*replacement*), and how the evolution should progress (the overall algorithm).

The genetic algorithm includes some representation-specific operators and some representation-neutral operators. The initialization, mating (typically implemented as *crossover*), and mutation operators are specific to the representation. Selection, replacement, and termination are all independent of the representation. In the context of the scheduling problem, the representation is specific to scheduling, but a general representation may be used with many different variations of the scheduling problem.

In traditional schedule optimization methods, the search algorithm is tightly coupled to the schedule generator. These methods operate in the problem space; they require information about the schedule in order to search for better schedules. Genetic algorithms operate in the representation space. They care only about the structure of a solution, not about what that structure represents. The performance of each solution is the only information the genetic algorithm needs to guide its search. For example, a typical heuristic scheduler requires information about the resources and constraints in order to decide which task should be scheduled next in order to build the schedule. The genetic algorithm, on the other hand, only needs to know how 'good' a schedule is and how to combine two schedules to form another schedule.

Having said that, many hybrid genetic algorithms exist which combine hill-climbing, repair, and other techniques which link the search to a specific problem space.

Proper choice of representation and tailoring of genetic operators is critical to the performance of a genetic algorithm. Although the genetic algorithm actually controls selection and mating, the representation and genetic operators determine *how* these actions will take place. Many genetic algorithms appear to be more robust than they actually are only because they are applied to relatively easy problems. When applied to problems whose search space is very large and where the ratio of the number of feasible solutions to the number of infeasible solutions is low, care must be taken to properly define the representation, operators, and objective function, otherwise the genetic algorithm will perform no better than a random search.

For example, in the solution of a continuous function of two variables, Grüninger showed that a real number representation consistently out-performs a binary-to-decimal encoding [Grüninger 96]. In addition, small changes to the algorithm and genetic operators had a significant impact on the algorithm's performance. Grüninger showed that the genetic operators must effectively balance exploration and exploitation so that the genetic algorithm will be able to both avoid local minima/maxima in global search and find small improvements in local search.

Some genetic algorithms introduce another operator to measure similarity between solutions in order to maintain clusters of similar solutions. By maintaining diversity in the population, the algorithms have a better chance of exploring the search space and avoid a common problem of genetic algorithms, *premature convergence*. After a population has evolved, all of the individuals typically end up with the same genetic composition; the individuals have *converged* to the same structure. If the optimum has not been found, then the convergence is, by definition, premature. In most cases, further improvement is unlikely once the population has converged.

The similarity measure is often referred to as a *distance function*, and these genetic algorithms are referred to as *speciating* or *niching* genetic algorithms. The similarity measure may be based upon the data in the genome (genotype-based similarity), it may be based upon the genome after it has been transformed into the problem space (phenotype-based similarity), or it may integrate some combination of these.

Sections 4.2.1-4.2.3 describe three variations of genetic algorithms: simple, steady-state, and struggle. The tests describe in Chapter 5 used the steady-state and struggle genetic algorithms.

The simple genetic algorithm is one of the more common genetic algorithm implementations; a description of this algorithm is included for clarity and comparison. The steady-state algorithm is another standard genetic algorithm made popular by the GENITOR program. The struggle genetic algorithm is a new kind of speciating genetic algorithm developed by Thomas Grüninger.

4.2.1 Simple Genetic Algorithm (Non-Overlapping Populations)

The simple genetic algorithm uses non-overlapping populations. In each generation, the entire population is replaced with new individuals. This process is illustrated in Figure 7. Typically the best individual is carried over from one generation to the next (this is referred to as elitism) so that the algorithm does not inadvertently forget the best that it found. Maintaining the best individual also causes the algorithm to converge more quickly; in many selection algorithms, the best individual is more likely to be selected for mating.

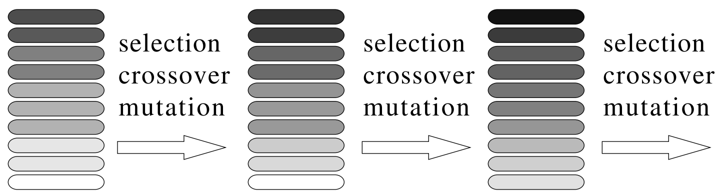


Figure 7 The "simple" genetic algorithm. This algorithm uses non-overlapping populations; the entire population is replaced each generation. Individual solutions are represented by shaded ovals. In this case, darker shading represents a better solution.

Since the entire population is replaced each generation, the only 'memory' the algorithm has is from the performance of the crossover operator. If the crossover accurately conveys good genetic material from parents to offspring, the population will improve. If the crossover operator does not maintain genetic material, the population will not improve and the genetic algorithm will perform no better than a random search. A crossover operator that generates children that are more often unlike their parents than like them leads the algorithm to do more exploration than exploitation of the search space. In search spaces with many infeasible solutions, such scattering will more often generate infeasible rather than feasible solutions.

4.2.2 Steady-State Genetic Algorithm (Overlapping Populations)

The steady-state genetic algorithm uses overlapping populations. In each generation, a portion of the population is replaced by the newly generated individuals. This process is illustrated in Figure 8. At one extreme, only one or two individuals may be replaced each generation (close to 100% overlap). At the other extreme, the steady-state algorithm becomes a simple genetic algorithm when the entire population is replaced (0% overlap).

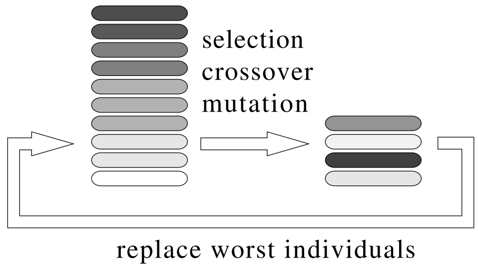


Figure 8 The "steady-state" genetic algorithm. This algorithm uses overlapping populations; only a portion of the population is replaced each generation. The amount of overlap (percentage of population that is replaced) may be specified when tuning the genetic algorithm.

Since the algorithm only replaces a portion of the population of each generation, the best individuals are more likely to be selected and the population quickly converges to a single individual. As a result, the steady-state algorithm often converges prematurely to a sub-optimal solution. Once again, the crossover and mutation operators are key to the algorithm performance; a crossover operator that generates children unlike their parents and/or a high mutation rate can delay the convergence.

4.2.3 Struggle Genetic Algorithm

The struggle genetic algorithm is similar to the steady-state genetic algorithm. However, rather than replacing the worst individual, a new individual replaces the individual most similar to it, but only if the new individual has a score better than that of the one to which it is most similar. This requires the definition of a measure of similarity (often referred to as a *distance function*). The similarity measure indicates how different two individuals are, either in terms of their actual structure (the genotype) or of their characteristics in the problem-space (the phenotype).

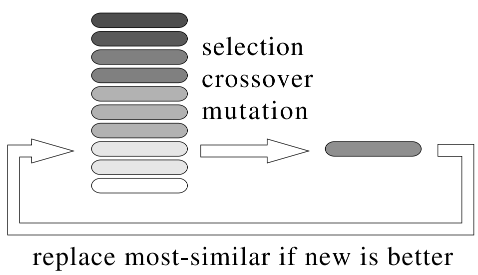


Figure 9 The "struggle" genetic algorithm. This algorithm is similar to the steady-state algorithm, but whereas the steady-state algorithm uses a "replace worst" strategy for inserting new individuals into the population, the struggle algorithm uses a form of "replace most similar".

The struggle genetic algorithm was developed by Grüninger in order to adaptively maintain diversity among solutions [Grüninger 96]. As noted previously, if allowed to evolve long enough, both the simple and the steady-state algorithms converge to a single solution; eventually the population consists of many copies of the same individual. Once the population converges in this manner, mutation is the only source of additional change. Conversely, a population evolving with a struggle algorithm maintains different solutions (as defined by the similarity measure) long after a simple or steady-state algorithm would have converged. Unlike other niching methods such as sharing or crowding [Goldberg and Richardson 87] [Mahfoud 95][De Jong 75], the struggle algorithm requires no niching radius or other parameters to tune the speciation performance. The struggle algorithm is similar to deterministic crowding and shares some characteristics of restricted tournament selection.

If the similarity function is properly defined, the struggle algorithm maintains diversity extremely well. However, like the other genetic algorithms, performance is tightly coupled to the genetic operators For example, if the crossover operator has a very low probability of generating good individuals when mating between or across species (as defined by the similarity measure), the algorithm will fail. If the mutation rate is too high, the algorithm will perform only as well as a random search.

4.3 Genetic Representation

Although much of the early genetic algorithm literature in the United States has focused on bit representations (i.e. solutions were encoded as a series of 1s and 0s), genetic algorithms can operate on any date type. In fact, most recent scheduling implementations use list-based representations. But whether the representation is a string of bits or a tree of instructions, any representation must have appropriate genetic operators defined for it. The representation determines the bounds of the search space, but the operators determine how the space can be traversed.

For any genetic algorithm, the representation should be a minimal, complete expression of a solution to the problem. A minimal representation contains only the information needed to represent a solution to the problem. A complete representation contains enough information to represent any solution to the problem. If a representation contains more information than is needed to uniquely identify solutions to the problem, the search space will be larger than necessary.

Whenever possible, the representation should not be able to represent infeasible solutions. If a genome can represent an infeasible solution, care must be taken in the objective function to give partial credit to the genome for its “good” genetic material while sufficiently penalizing it for being infeasible. In general, it is much more desirable to design a representation that can only represent feasible solutions so that the objective function measures only optimality, not feasibility. A representation that includes infeasibles increases the size of the search space and thus makes the search more difficult.

The following representation for scheduling is a minimal representation that can represent resource-infeasible solutions. As shown in Figure 10, a genome consists of an array of relative start times and an array of integer execution modes for each task. Each time represents the duration from the latest finish of all predecessor tasks to the start time of the corresponding task. Each mode represents which of the possible execution modes will be used for the corresponding task. As shown in the figure, the modes are typically defined in terms of resource requirements. This representation is *not* order-based. The elements in the array correspond to the tasks in the work order or project plan, but the order of elements relative to each other is insignificant. Each genome is a complete schedule; the genome directly represents a schedule by encoding both start times (explicitly) and resource assignments (via the execution mode).

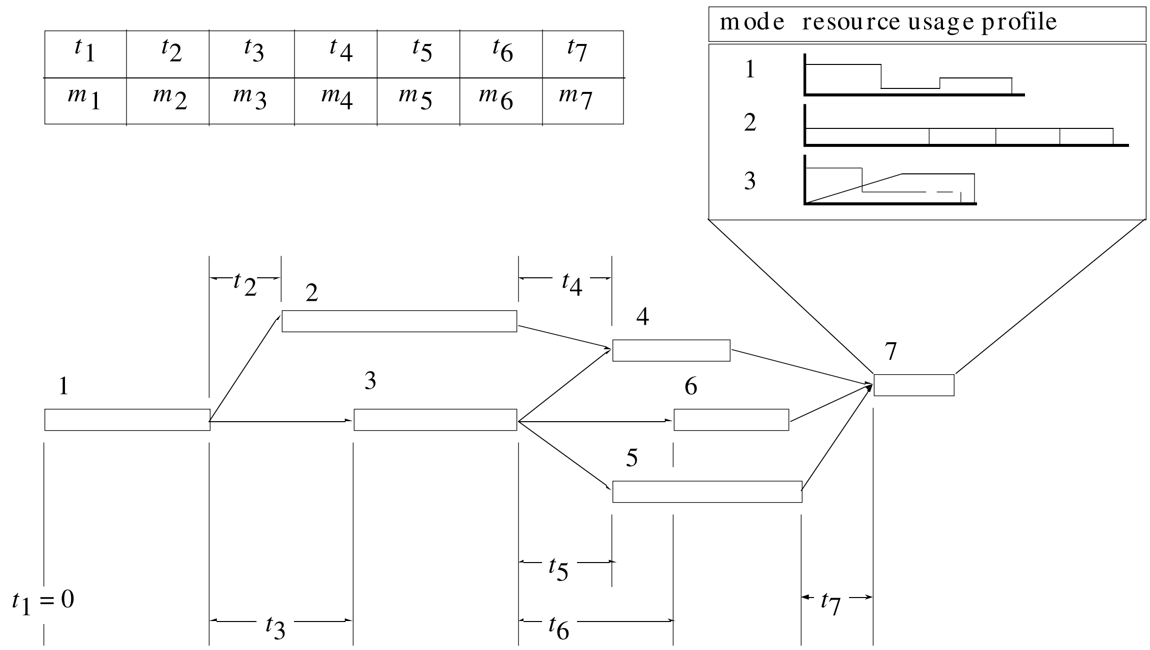


Figure 10 The genome and its mapping to the schedule. A single genome is a double array of floating-point start times and integer execution modes. Each element in the arrays corresponds to a task in the project plan or work order. The times represent delay times relative to the estimated finish time of the predecessors. The execution modes vary from task to task and represent one of the possible execution modes for the corresponding task.

This representation assumes that the plan exists; execution modes for each task must be completely defined and the constraints and objectives must be defined *a priori*. The genome is not a representation for evolving plans, but it can select between plans in the form of multiple execution modes for individual tasks. In other words, the topology of the project (the set of precedence relationships between tasks) does not evolve. In addition, the abstraction from task to time-mode pairs permits the genome to *adapt* to changes in the plan. New modes can be defined or existing modes can be modified without requiring a reformulation of the problem.

4.3.1 Start Times

A single genome contains an array of real numbers. Each real number is associated with a task and indicates the period of time from the latest finish of all the task's predecessors to the start of the task. The times are measured relative to other tasks, not to an absolute reference. If the precedence constraints for a task allow overlap, then negative values are permitted for the start time for that task. If overlap is not modeled or not permitted, values are truncated to zero. As a result, precedence feasibility is assured for every schedule.

4.3.2 Operating Modes

Each genome contains an array of integers which represent execution modes. Each index indicates in which mode its corresponding task will be executed. If a task has been defined with multiple modes of operation, the corresponding integer value may assume a value equal to one of the defined modes. If a task has only one execution mode, its corresponding execution mode indicator will never change.

4.3.3 Additional Characteristics

The combination of relative start times and mode indices completely and uniquely define a schedule. Given a genome, the corresponding schedule can be evaluated to see if resources are violated and/or to see how well/poorly the objectives are met. Precedence relationships are always enforced; only resource and/or temporal constraints may be violated. In fact, temporal constraints can be enforced by specializing the genetic operators. For example, crossover may force a task to begin at a certain time, or may assign a resource in a certain amount for a given period. The genetic operators reinforce constraint satisfaction whenever possible so that precedence-infeasible solutions are impossible to generate. As a result, the algorithm searches only in the realm of resource feasibility rather than also in the realm of precedence feasibility.

This representation supports the modeling assumptions outlined in Section 4.1. Of particular note is the ease with which various objectives and constraints can be included and/or modified. Since each genome represents a complete schedule, almost any objective measure can be used. The objective measure is not entwined in either the search algorithm or the representation.

The genetic algorithm operates on the structure of the genome without considering what the genome means in the schedule space. As a result, the constraints are coupled to the search algorithm only through the representation-specific genetic operators (initialization, crossover, and mutation), and the objective measures are completely decoupled from the search algorithm. Only the initialization, crossover, and mutation operators are tightly coupled to the representation, and these operators are loosely coupled to the constraint models (and then only for some problem instances).

4.4 Genetic Operators

Use of a genetic algorithm requires the definition of initialization, crossover, and mutation operators specific to the data type in the genome. In addition, a comparison operator must also be defined for use with niching/speciating genetic algorithms such as the struggle genetic algorithm.

4.4.1 Initialization

The real number part of the genome was initialized with random numbers. The range of possible values was based upon the average estimated task durations. The magnitude of the numbers matters because the algorithm finds better solutions faster if the random numbers are the same order of magnitude as the task durations.

The mode for each task was randomly selected from the set of modes available for that task. No task could be assigned a mode which was not defined. Note that the tasks in a given project plan did not necessarily have the same number of modes.

4.4.2 Crossover

The crossover operator included two parts, one for each data type in the genome. Blend crossover, a real-number-based operator, was used for the array of time values. Uniform crossover, a type-independent operator, was used for the array of execution modes.

Blend crossover was used to generate new values for each element in the time component of the genome. As illustrated in Figure 11(a) and described by Eshelman and Schaffer, the blend crossover generates a new value depending upon the distance between parent values [Eshelman and Schaffer 1992]. Blend crossover is adaptive and needs no tuning parameter; when the parent values are further apart, the child's value may end up further from the parents than it would if the parents were closer together.

Three other crossover methods for the time component of the genome were implemented as illustrated in Figure 11(b-d). These include (b) mean with Gaussian noise, (c) extrapolation, and

(d) uniform. In cursory testing prior to the longer runs, uniform crossover performed slightly worse than the other methods, but in general, the effects on algorithm performance from changing crossover operators were overshadowed by the effects of changes to the search algorithm and variations in problem structure. Blend crossover was used for the tests described in Chapter 5.

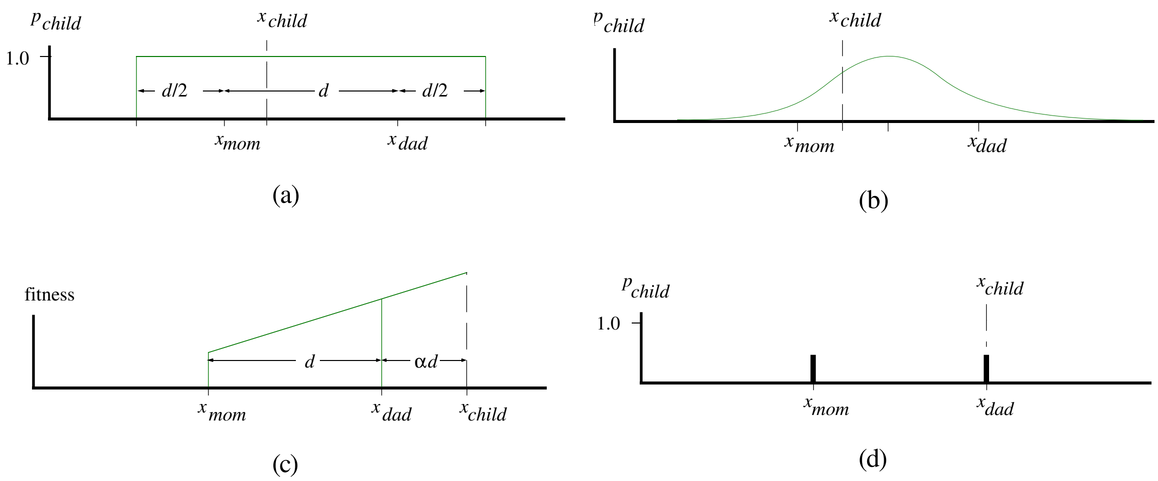


Figure 11 Four different crossover operators for generating a new time value from two parent values. Blend crossover (a) adaptively selects a value for the child based on the distance between the parents. Mean-with-noise crossover (b) generates a child value based on the mean of the parent values, then adds a random number. The extrapolation crossover (c) uses the objective scores of the parents to project a value for the child and choose a value based on that projection. Uniform crossover (d) sets the child value by picking randomly from the two parent values. Blend crossover was used in the experiments in Chapter 5.

The execution modes were selected by randomly choosing from one parent or the other as shown in Figure 12. Uniform crossover operates independently of the data type; each element is copied from one parent or the other. If two children are generated in the mating, then if one child inherits an element from the mother, the other child inherits the corresponding element from the father, and *vice versa*.

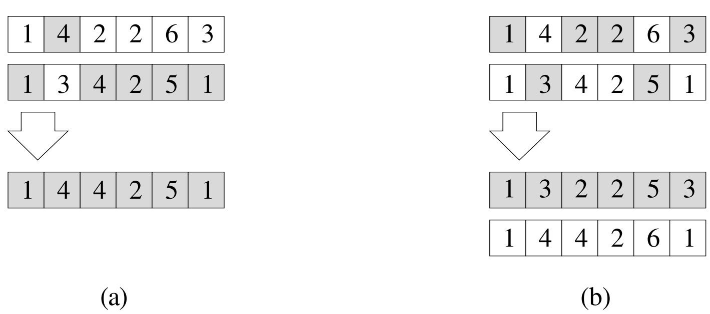


Figure 12 Uniform crossover. If one child is produced in the mating (a), each of its elements are inherited randomly from one parent or the other. If two children are produced in the mating (b), an element inherited in one child from the mother is inherited in the other child from the father.

Time values in the child genome were truncated as needed. For example, if a task did not allow overlap, the value was truncated to zero. This guaranteed precedence-feasible solutions.

Note that all of the crossover techniques listed here required two parents (sexual crossover) but were capable of generating either one or two children.

A crossover probability was included in the steady-state genetic algorithm. A probability of 1.0 meant that the new individuals would be created from a mix of genetic material from both parents. A probability less than 1.0 meant that, in some cases, the genetic material from the parents could be copied directly into the new individuals with no crossing over.

4.4.3 Mutation

Mutation was performed by applying Gaussian noise to each element in the real number array and by flipping modes in the mode array.

As illustrated in Figure 13, a single element in the genome was mutated by replacing it with a number chosen based upon a Gaussian curve defined by a mean and standard deviation. The mean is equal to the previous value. The deviation should be adaptive, but in the tests reported in this thesis, the deviation used to define the Gaussian curve was fixed.

Execution modes were mutated by randomly picking between valid execution modes. If an element in the mode array was selected to be mutated, its value was changed to another valid mode from its corresponding task.

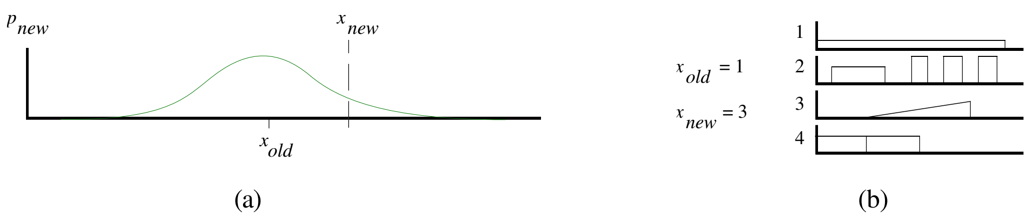


Figure 13 The mutation operators. If selected for mutation, a time was mutated by applying Gaussian noise to the existing value (a). An execution mode was mutated by replacing the current mode with a different mode from the set of modes for the corresponding task (b).

If task overlap was included in the task models, then negative delay times were permitted. If overlap was not allowed, delay times were truncated to zero so that no precedence-infeasible solutions could be generated.

The mutation probability was gene-based, not genome-based. For example, if a mutation probability of 50% was specified, then each activity start time had a 50% chance of being mutated. Mutation of a given start time did not require a corresponding mutation of execution mode; the mutation operated on the times independently from the execution modes.

4.4.4 Similarity Measure

The similarity function compares two solutions and returns a value that indicates how much the solutions differ. Often called a 'distance' function, this operator is typically used by speciating genetic algorithms. Many different similarity measures can be defined for any given representation. This section describes two similarity measures for the scheduling genome: a distance-based measure (Euclidean) and a sequence-based measure (Sequence). Both of these similarity measures neglect the mode components of the genome.

Euclidean. As illustrated in Figure 14, the distance-based, or “Euclidean” comparator uses the square root of the sum of the squares of the “distance” between each element in the genome. Note that the Euclidean comparator can be implemented in at least two ways. In the first implementation, the distances are calculated directly from the relative times in the genome. In the second variation, the times are transformed to absolute times before the distance is calculated.

$$d = \sqrt{\sum_{i=1}^{n} \left(a_i - b_i\right)^2}$$

a_i - ith element of genome A

b_i - ith element of genome B

n - number of tasks

(a)

A: | 1.5 | 2 | 1 |

B: | 2.5 | 3 | 1.1 |

$$d = \sqrt{(1.5-2.5)^2 + (2-3)^2 + (1-1.1)^2}$$

(b)

Figure 14 The Euclidean similarity measure. The similarity, d, of two genomes is based upon the Euclidean distance between the task start times as shown in (a). The example in (b) illustrates the comparison of two genomes, each with three elements. For clarity, the mode components are not shown.

Sequence. In this method, two solutions are compared based on the absolute order in which tasks are initiated. Figure 15 illustrates the sequence-based comparator. The comparison is made by first generating the sequence of activities based on the absolute start time of each task. If multiple tasks share the same start time, they share the same position in the sequence. Next, a square incidence matrix is generated with number of rows equal to the number of tasks. The matrix is initially populated with zeros, then ones are added based on the sequence. A one is inserted in cell i,j if task j follows task i. This procedure is then applied to the second genome to generate a second matrix. The comparison score is found by adding the number of cells that differ between matrices. Two identical schedules will have a lower score than two schedules with different sequences.

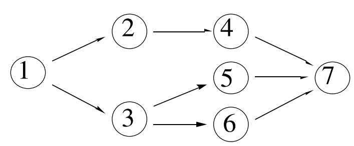
$$d = \sum_{i=1}^{n} \sum_{j=1}^{n} \left(a_{i,j} \neq b_{i,j} \right)$$

$a_{i,j}$ - element *i,j* from A's matrix

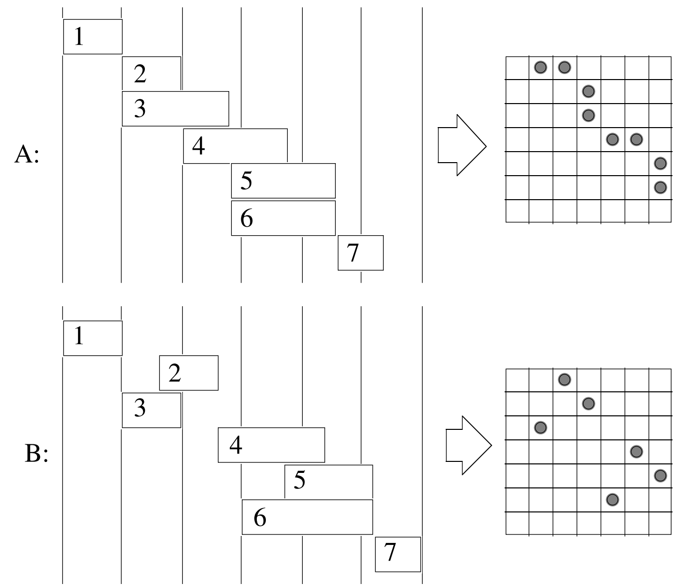
$b_{i,j}$ - element *i,j* from B's matrix

n - number of tasks

(a)



(b)



(c)

Figure 15 Calculation of the sequence similarity measure. The similarity, *d*, of two genomes is based on the upon the order of tasks that the genomes share as shown in (a). A sample comparison is shown in (b) and (c). The precedence relations are shown in (b). Two feasible schedules and their corresponding incidence matrices are shown in (c). The start times of each task are used to determine the absolute sequence of tasks, then this sequence is used to populate an incidence matrix. The similarity is calculated by counting the number of cells in the incidence matrix that are the same. In the example shown here, the similarity measure is 6.

The sequence similarity method is similar in some respects to the edge recombination operator defined by Whitley for use with the traveling salesman problem [Whitley et al 1989]. Note, however, that whereas Whitley used the incidence matrix for determining how crossover should be executed, here it is used only for determining similarity.

4.5 Objective Function

The genome performance measure, often referred to as the *objective function*, consists of two parts, each based upon the schedule the genome represents. The first part is a measure of constraint satisfaction, the second part is based on the schedule performance with respect to the objectives. Since the genome directly represents a schedule, calculation of both measures is straightforward. Some typical constraint and objective measures are outlined in this section, followed by an explanation of how the constraint and objective measures were combined to produce the overall score for each genome.

4.5.1 Constraints

Most measurements of constraint satisfaction were based upon *resource profiles*. Resource profiles define resource availability or consumption as a function of time.

4.5.1.1 Resource Availability

Part of the planning stage is the definition of resource availability. For each resource, a profile of availability can be generated to indicate when and how much of that resource will be available. Sample resource availability and requirement profiles are illustrated in Figure 16.

Note that this representation encompasses both resource quantity and temporal restrictions on resource usage.

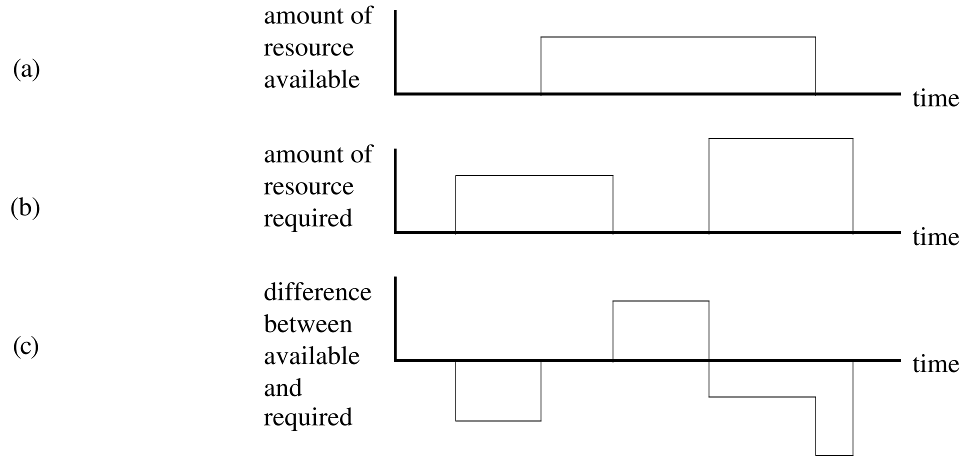


Figure 16 Resource availability, requirement, and feasibility profiles. (a) represents resource availability as a function of time, (b) shows the resource requirements as a function of time, and (c) is the result of subtracting the requirements from the available resources. Any negative segments in the difference (c) indicate that the schedule is not feasible because more resources are required than are available.

The feasibility of a given schedule with respect to a given resource is calculated by comparing the resource availability curve to the resource requirement curve for the resource in question. The feasibility score is the integral of the availability-required difference with respect to time for all intervals in which the difference is less than zero. In the example shown in Figure 16(c), the feasibility score is equal to the sum of the areas of the two regions that dip below the time axis. This measure of feasibility results in a graduated score that reflects not only whether or not a solution is feasible, but also the degree of infeasibility if it is infeasible.

4.5.1.2 Temporal Constraints

If a task *must* be started at a specific time, then the corresponding start time in the genome is adjusted by the genetic operators so that the task always starts at that time. If a resource is available only at certain times or for a certain duration, this is reflected in the construction of the availability profile for that resource.

4.5.1.3 Precedence Feasibility

Precedence feasibility is enforced by the representation and genetic operators, so precedence-infeasible solutions are not possible.

4.5.2 Objectives

Many different measures of schedule performance exist. The representation described in Section 4.3 permits modification of objective measures with little or no effect on the search algorithm or genetic representation. The next three sections highlight some of the more common performance measures.

4.5.2.1 Due Dates and Tardiness

The performance of many projects is measured in terms of due dates or deviation from projected finish times. These measures are calculated directly from the schedule. For example, if a work order specifies that 80% of the jobs must be completed by their specified finish times, the performance measure can be calculated directly. If each job has a due date, x_i, specified in the plan and finish time, f_i, determined from the schedule, the tardiness is the difference $d = f_i - x_i$ where d is truncated to zero (early jobs are not tardy). The mean tardiness for a work order is simply the average of the tardiness scores of each job.

4.5.2.2 Cost

The total cost of a schedule can be found by adding the individual costs of each activity given the execution mode and resources applied to it. Since the schedule is explicitly defined, any genome can be used to calculate a net-present value or virtually any other cost measurement of performance. If each task has a cost, c_i, determined from the scheduled modes, then the total cost is simply the sum of the costs of each task.

4.5.2.3 Makespan

The length of time required to complete a schedule is calculated directly from the information in the genome. The makespan is simply the finish time of the last task. Note that a schedule may indicate a makespan when, in fact, that schedule is infeasible due to violations of resource constraints.

4.5.3 Composite Scoring

The score for any genome consists of two parts: a constraint satisfaction part and an objective performance part. Since the objective measures are, in practice, meaningless if the schedule is infeasible, none of the objectives are considered until all of the constraints have been satisfied. The degree to which constraints are violated determines how feasible the schedule is, and if the schedule is feasible the objective performance is then considered. For a genome with constraint performance p_c and objective performance p_{obj}, the overall score is shown in Equation 1. Details about the calculation of constraint and objective performance are described in the next two sections.

$$\text{score} = \left[\begin{array}{ll} \dfrac{p_c}{2} & \text{if } p_c < 1 \\ \dfrac{p_c + p_{obj}}{2} & \text{if } p_c = 1 \end{array} \right. \qquad \textbf{Equation 1}$$

The score for each genome ranges from 0 to 1, inclusive. Any genome with a score less than 0.5 does not satisfy all of the constraints and is thus infeasible. Any genome with a score of 1.0 not only satisfies all constraints but also has a perfect score with respect to each objective measure.

When objectives and constraints are treated separately, the genetic algorithm implicitly distinguishes between feasible and infeasible solutions. The threshold before considering objectives is critical; if a single score value can represent both an infeasible solution with good objective performance and a feasible solution with poor objective performance, the genetic algorithm may be deceived and end up favoring infeasible solutions with better objective scores.

Numerous experiments were run in which constraints and objectives were given equal weight in the calculation of the genome score. In many cases, an infeasible solution was found whose strong objective score belied its infeasibility. Since the genetic algorithm had only the overall score on which to base its selection, the infeasible solutions often dominated the evolution. The searches often resulted in infeasible schedules, even when better, feasible solutions were known to exist.

4.5.3.1 Constraint Satisfaction Part

Each schedule contains multiple constraints, each of which measures some aspect of the feasibility of the schedule. For each constraint, i, a measure of constraint violation, x_i, was defined. For resource availability, the constraint violation measure was equal to the difference between the resources available and the resources required. Temporal constraints were typically measured based on the variance between actual times and desired times.

The constraint measures were normalized to a scale from 0 to 1, inclusive, using a specifications-based transformation such as those illustrated in Figure 17. The raw constraint measure was transformed to the interval [0,1] in order to facilitate comparison of multiple objectives. The transformations allow specification of feasible ranges as well as single values, and they transform constraint measures in various sets of units to the same zero-to-one, unitless interval. Also, as described by David Wallace, this method of transformation permits an explicit clarification of desired performance [Wallace 95].

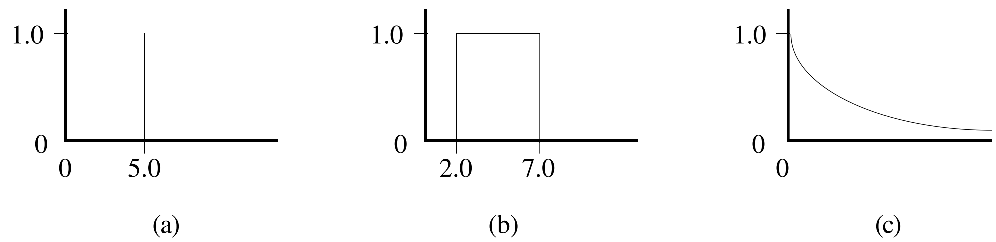


Figure 17 Transformation of constraint components to the interval [0,1]. (a) shows a specification in which only a value of 5 is acceptable, the specification in (b) accepts any value between 2 and 7, and (c) shows a specification in which a value of 0 is acceptable and any values greater than zero receive progressively worse scores.

The performance, g_i, of the ith constraint is a function of the measure of constraint violation, x_i. For evaluating resource constraints, the transformation in Equation 2 was used. This transformation gives most credit to solutions which satisfy all of the constraints, but gives some credit to those that partially satisfy the constraints.

$$g_i = \frac{1}{1+x_i} \qquad \textbf{Equation 2}$$

The composite constraint satisfaction measure, p_c, is calculated by averaging the components from n transformed constraint measures as shown in Equation 3. The components were averaged rather than multiplied so that partial credit would be given for infeasible solutions. If the components had been multiplied, one component with a score of zero would have annihilated all the others. Although annihilation is philosophically the ‘right’ approach since failure of any constraint means that no other constraint matters, in practice the genetic algorithm needs *some* feedback about the degree of infeasibility on which to base its search.

$$p_c = \frac{1}{n}\sum_{i=1}^{n} g_i \qquad \textbf{Equation 3}$$

In a more general approach, the transformed constraint scores could be weighted to indicate differences in importance between constraints. However, this such an approach would require the definition of arbitrary weighting factors. In addition, the constraints are implicitly equal in importance, so assigning degrees of importance would be meaningless. The implicit equality stems from the fact that all constraints must be satisfied before the optimality is measured; if one constraint fails, they all fail.

The complete process of calculating a constraint satisfaction measure is illustrated in Figure 18. A project with the three constraint measures in the figure has a composite constraint score, p_c, of 0.492. In this example, the individual components, x_1, x_2, and x_3, are the constraint feasibility measures for resource types 1, 2, and 3, respectively. For non-renewable resources such as resource type 3, an arbitrary cutoff time was defined to enable calculation of the constraint violation. The cutoff is required in this case since the non-renewable resource has no explicit time at which it is no longer available other than the time at which it is completely consumed. If the resource were not consumed, it would persist indefinitely.

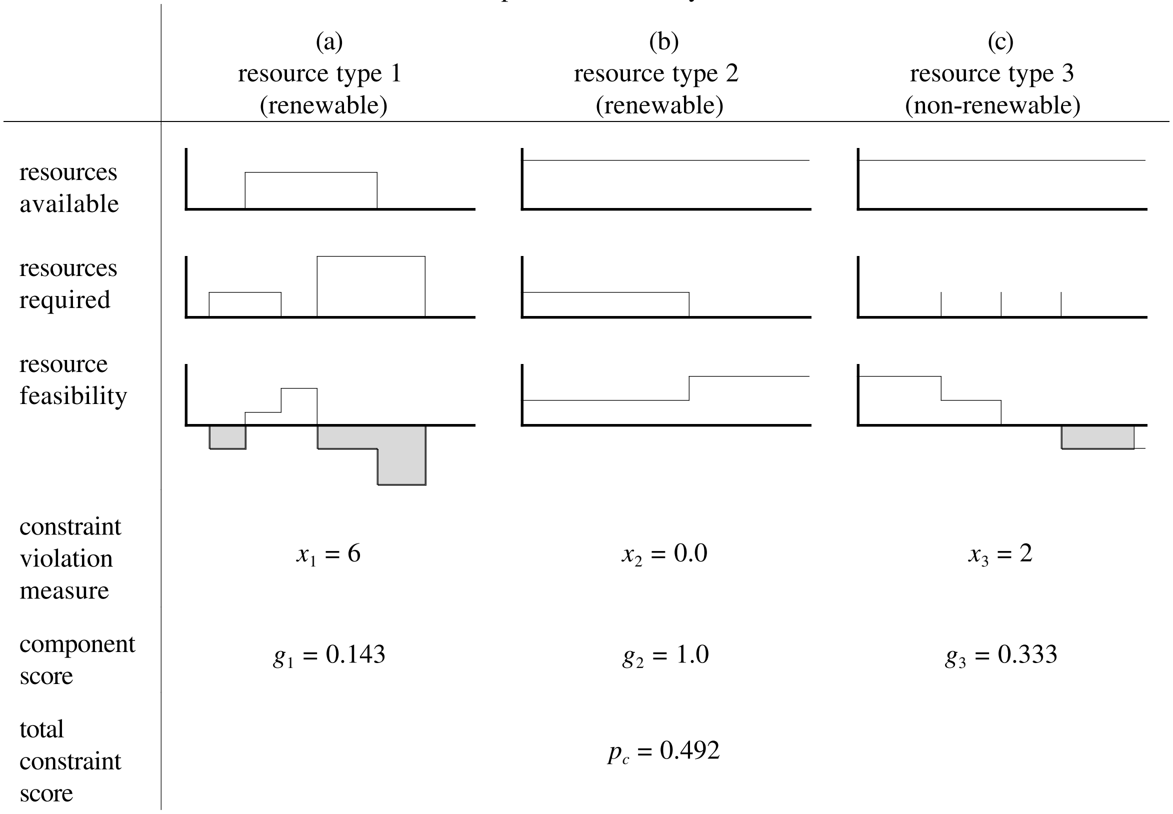


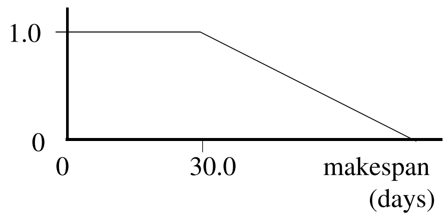
Figure 18 Example of constraint satisfaction measure calculation. The project in this example has three resource types with constraints defined by the resource availabilities and requirements defined by the profiles in (a), (b), and (c). The first two are renewable resources, the third is nonrenewable. Subtracting the usage from the availability profiles yields the actual constraint satisfaction curves. The score from each of these curves is then normalized using the transformation in Equation 2. The final constraint score is the average of the component scores.

Giving credit for partially feasible schedules is critical. Many schedules may be infeasible, but many infeasible schedules have feasible portions. One option for evaluating the score of a genome is to give it a score of 0 if it does not meet the feasibility requirements. However, a score of 0 provides no feedback to the genetic algorithm about the value of one genome over another, so a genetic search among infeasibles can be no better than a random search. On the other hand, when a graduated score for infeasible solutions is used, the genetic algorithm can evolve solutions that are more feasible and may eventually find solutions that are completely feasible.

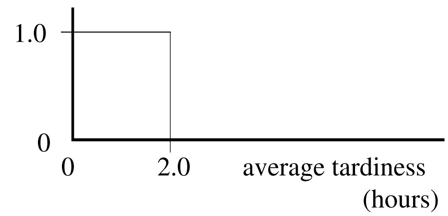
Note that the question of feasibility varies from problem to problem and depends upon the formulation of the constraints and objective measures. One problem may have a single feasible solution, and thus one optimal solution. Another may have many feasible solutions but only a single optimal solution. Yet another may have many feasible solutions and nearly as many optimal solutions.

4.5.3.2 Objective Performance Part

A project may have a single objective or multiple, possibly conflicting, objectives. Each objective is normalized then the lot are averaged to form the overall objective performance, p_{obj}. Each objective is normalized to a scale from 0 to 1, inclusive, where 1 indicates perfect satisfaction of the objective measure. The normalization is done using the specifications-based transformations described in the previous section. Typical transformations are illustrated in Figure 19.



(a)



(b)

Figure 19 Transformations from raw measure of performance to normalized component score. (a) shows a transform for makespan in which any makespan over 30 days is permissible but not desired; longer makespans are less desirable than shorter makespans. (b) shows a transform for average tardiness in which the average tardiness for the work order must be less than 2 hours.

The transformed score, f_i, of the ith objective is a function of the raw objective measure, y_i. The transformed components are averaged to form the final objective measure, p_{obj}, as shown in Equation 4.

$$p_{obj} = \frac{1}{m}\sum_{i=1}^{m} f_i \qquad \textbf{Equation 4}$$

If some objectives are more important than others, weights may be applied to the transformed scores as shown in Equation 5. Each weight, w_i, indicates the value of the ith objective relative to the other objectives.

$$p_{obj} = \frac{\sum_{i=1}^{m} w_i f_i}{\sum_{i=1}^{m} w_i} \qquad \textbf{Equation 5}$$

Note that averaging the components may result in dominance, wherein a single large score overpowers other scores. Transforming the raw performance measures to the [0,1] interval mitigates the effects of dominance to some extent, but the transformations must be defined so that changes in the raw objective measure are adequately reflected in the transformed score. An alternative method of combining components is multiplication, but a score derived by multiplication may suffer from annihilation; when the components are multiplied, a score of 0 on any single component causes all the components to be 0.

5. Test Problems and Results

5.1 The Test Problems

The genetic algorithm was run on the following sets of test problems:

- Patterson's project scheduling problems (PAT)
- single mode project scheduling set by Kolisch *et al* (SMCP)
- single-mode full-factorial set by Kolisch *et al* (SMFF)
- multi-mode full-factorial set by Kolisch *et al* (MMFF)
- job-shop problems from the operations research "warehouse" (JS)
- the benchmarx problems by Fox and Ringer (BMRX)

First introduced by James Patterson in his comparison of exact solution methods for resource-constrained project scheduling, the Patterson set (PAT) consists of 110 project scheduling problems whose tasks require multiple resources but are defined with only one execution mode.
The problems in the Patterson set are considered easy. First of all, with only 7-48 tasks per problem, the problems are not very big. Perhaps more importantly, the resource constraints are not very tight; in many cases the optimal resource-constrained solution is the same as the resource-unconstrained solution.

Kolisch described a method for generating project scheduling problems based on various parameters for controlling number of tasks, complexity of precedence relations, resource availability, and other measures [Kolisch 92]. The SMCP, SMFF, and MMFF problem sets were generated using *ProGen*, Kolisch's implementation of the algorithm he described.

The single mode set (SMCP) are similar to the Patterson set, but they range in size from 10 to 40 tasks and include more resource restrictions. The set includes 200 problems with 1 to 4 renewable resource types. Each task has only one execution mode.

The single mode full factorial set (SMFF) consists of 480 problems. Each problem has 30 tasks and 1 to 4 resource types, all renewable. Each task has only one execution mode. The set was generated by varying three parameters: network complexity, resource factor, and resource strength. These factors correspond roughly to the interconnectedness of the task dependencies, the number of resource *types* that are available, and resource *quantity* availability.

The multi-mode, full factorial set (MMFF) consists of 538 problems that are known to have feasible solutions from an original set of 640. The possibility of generating problems with no solution arises with the addition of non-renewable resources. The problems include four resource types, two renewable and two non-renewable. The number of activities per project is 10, and each activity has more than one execution mode. The set was generated by varying three parameters: network complexity, resource factor, and resource strength. Complete details of the problem generation are given in Kolisch description [Kolisch 92].

The jobshop problems (JS) are from the 'jobshop1' compilation of problems from the operations research library (http://mscmga.ms.ic.ac.uk/). The set consists of 82 problems commonly cited in the literature. The problems are the standard nxm jobshop formulation in which n jobs with m steps (tasks) are assigned to m machines (resources). They range in size from 6x6 to 15x20. In other words, they range from 36 tasks and 6 resources to 300 tasks and 20 resources. Each task has its own estimated duration, and each task must be performed by one (and only one) resource in a specific order. The objective of each problem is to minimize the makespan. Descriptions of the problems may be found in [Adams et al, 1988] [Fisher and Thompson, 1963] [Lawrence, 1984] [Applegate and Cook 1991] [Storer et al, 1992] [Yamada and Nakano, 1992].

The benchmarx problem was proposed by Barry Fox and Mark Ringer in early 1995. It is a single problem with 12 parts. Each part adds additional constraints or problem modifications that test various aspects of a solution method. The first four parts are fairly standard formulations. It gets harder from there. The problem is large: 575 tasks, 3 types of labor resources and 14 location-based resources. In addition to resource/location constraints, it includes many temporal restrictions such as three shifts per day with resources limited to certain shifts and task start/finish required within a shift or allowed to cross shifts. The last of the twelve parts includes multiple objectives. By varying resource availability and work orders after a schedule has been determined, the problem also tests the ability of solution methods to adapt to dynamic changes.

The characteristics of the problem sets are summarized in Table 3. With the exception of the last eight parts of the benchmarx problem, optimal solutions and best-known solutions are commonly available.

	number of problems	number of activities per problem	number of renewable resources per problem	number of non-renewable resources per problem	characteristics
PAT	110	7-48	1-3		project scheduling, single mode, multiple resources per task
SMCP	200	10-40	1-6		project scheduling, single mode, multiple resources per task
SMFF	480	30	4		project scheduling, single mode, multiple resources per task
MMFF	538	10	2	2	project scheduling, multi-mode, multiple resources per task
JS	82	36-300	6-20		job-shop scheduling, single mode, one resource per task
BMRX	1 (12)	575	17		general scheduling, one resource per task

Table 3 Characteristics of the test suites. Tasks in the project scheduling problems typically required more than one resource per task, whereas those in the job-shop problems required only one resource per task. All of the problems have feasible solutions. Optimal solutions are known for many of the problems, best-known solutions are used for comparison when no optimal solution is known.

Although the representation described in Chapter 4 supports multiple objectives, with the exception of the benchmarx problem, the objective for all of these problem sets was only to minimize the makespan. In addition, only the benchmarx problem specifies temporal constraints. All of the problems with renewable resources specify uniform resource availability, so a feasible solution is guaranteed for those problems. The multi-mode full factorial set includes non-renewable resources, so a feasible solution is not guaranteed for problems in this set. However, the published results of Sprecher and Drexl show optimal solutions for the 538 problems in the MMFF set [Sprecher & Drexl 96].

Most of the results were achieved using a steady-state genetic algorithm. However, some runs were made using the struggle genetic algorithm in order to evaluate the effects of speciation on the genetic algorithm performance on these problems.

No specific attempt was made to tune the genetic algorithm; it was run for a fixed number of generations with roulette wheel selection, a reasonable mutation rate, population size, and replacement rate. Table 4 summarizes the parameters used in the genetic algorithm runs in Figures 20-28.

	genetic algorithm	number of generations	mutation probability	crossover probability	population size	number of runs
PAT-SS-500-50	steady-state	500	3%	90%	50	50
PAT-SS-1000-50	steady-state	1000	3%	90%	50	10
SMCP-SS-500-50	steady-state	500	3%	90%	50	50
SMCP-SS-500-100	steady-state	500	3%	90%	100	10
SMFF-SS-500-50	steady-state	500	3%	90%	50	10
MMFF-SS-500-50	steady-state	500	3%	90%	50	20
MMFF-STR-500-50	struggle	500	3%	(100%)	50	10
MMFF-STR-500-50	struggle	500	3%	(100%)	50	10
JS-SS-2000-50	steady-state	2000	3%	90%	50	10

Table 4 Genetic algorithm parameters used for each set of tests. Crossover probability was specified for the steady-state algorithm since it may generate individuals by simply copying parents if no crossover is specified. The struggle algorithm always performs crossover.

The genetic algorithm required no modifications to switch between any of these problem sets. The benchmarx problem required additional data structures to include shift constraints and other modeling parameters, but no change to the algorithm or genome was required.

5.2 Genetic Algorithm Performance

Figures 20-28 summarize the performance of the genetic algorithm on the PAT, SMCP, SMFF, MMFF, and JS problem sets. In each figure, the results of the genetic algorithm are compared to the optimal score if it is known, or the published best if an optimal score is not known. In these problem sets, the performance measure is simply the makespan. The figures show the genetic algorithm performance relative to the best solution, so a value of 0% means that the genetic algorithm found the optimal makespan, a value of 100% means that the genetic algorithm found a makespan twice as long as the published best.

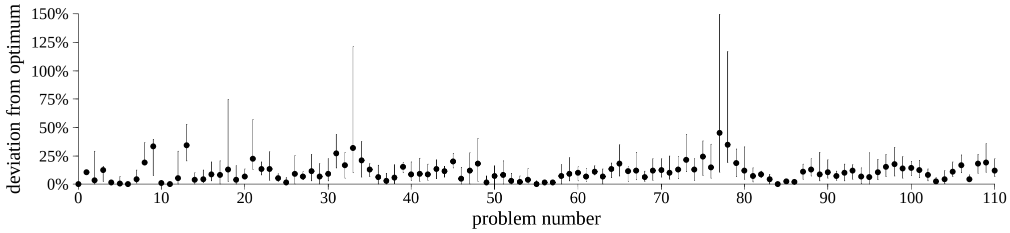


Figure 20 Summary of best, mean, and worst genetic algorithm performance on the Patterson problem set using a steady-state genetic algorithm for 500 generations with population size of 50 individuals (PAT-SS-500-50).

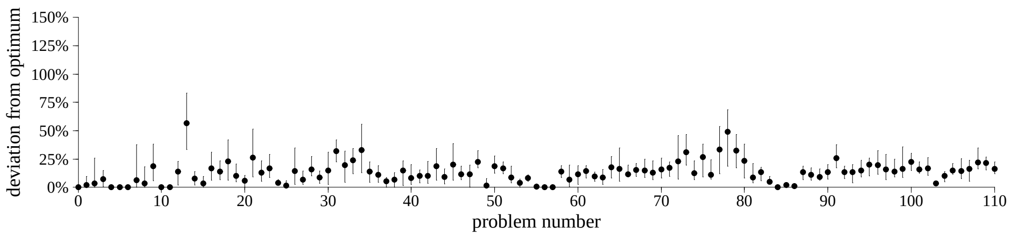


Figure 21 Summary of best, mean, and worst genetic algorithm performance on the Patterson problem set using a steady-state genetic algorithm for 1000 generations with population size of 50 individuals (PAT-SS-1000-50).

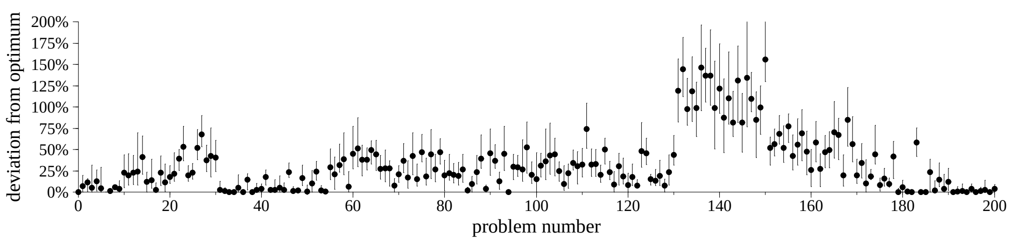


Figure 22 Summary of best, mean, and worst genetic algorithm performance on the single-mode project scheduling problem set using a steady-state genetic algorithm for 500 generations with a population size of 50 individuals (SMCP-SS-500-50).

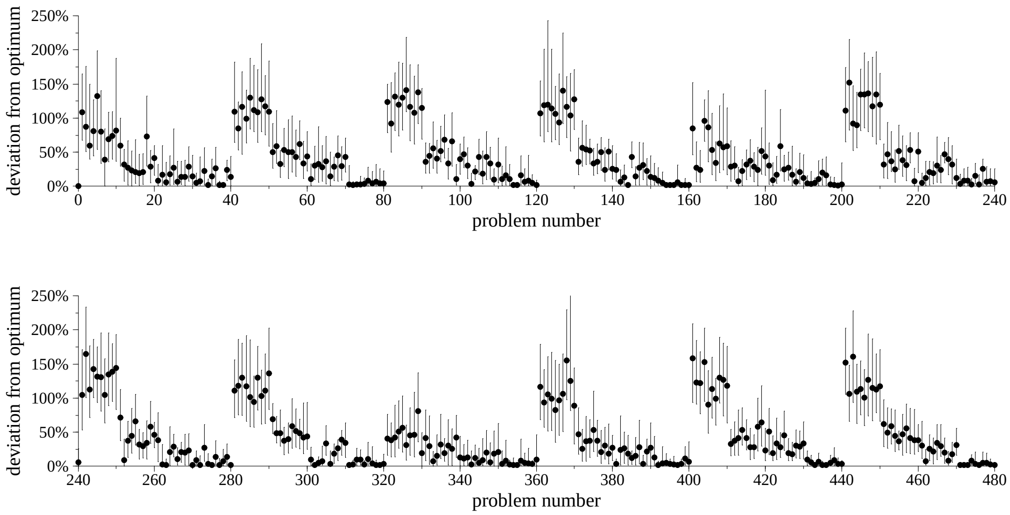


Figure 23 Summary of best, mean, and worst genetic algorithm performance on the single-mode full factorial problem set using a steady-state genetic algorithm for 500 generations with a population size of 50 individuals (SMFF-SS-500-50).

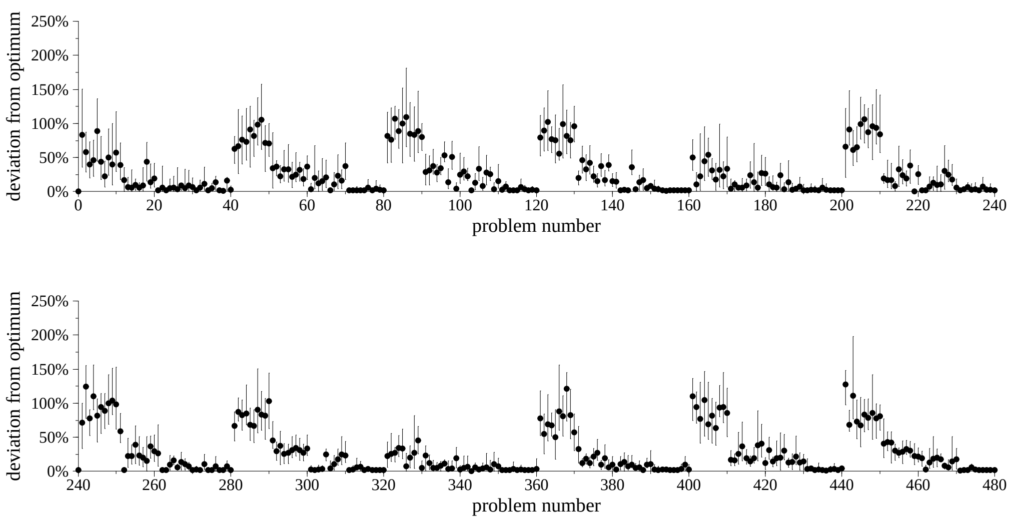


Figure 24 Summary of best, mean, and worst genetic algorithm performance on the single-mode full factorial problem set using a steady-state genetic algorithm for 1000 generations with a population size of 50 individuals (SMFF-SS-1000-50).

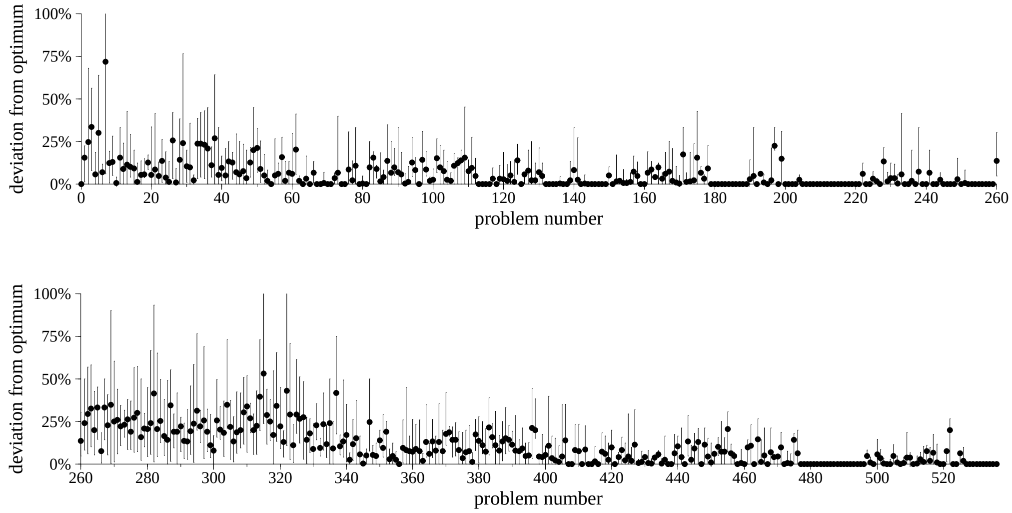


Figure 25 Summary of best, mean, and worst genetic algorithm performance on the multi-mode full factorial problem set using a steady-state genetic algorithm for 500 generations with a population size of 50 individuals (MMFF-SS-500-50).

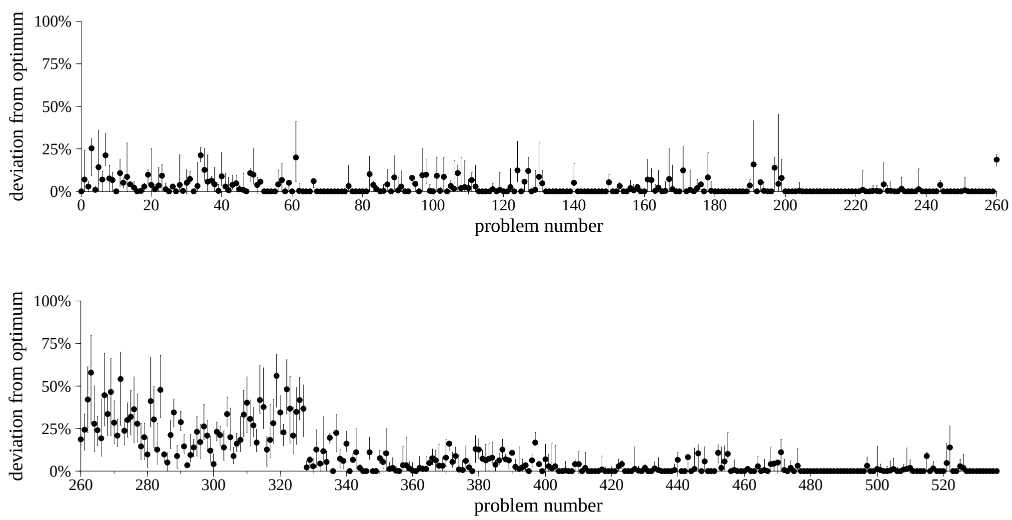


Figure 26 Summary of best, mean, and worst genetic algorithm performance on the multi-mode full factorial problem set using a steady-state genetic algorithm for 500 generations with a population size of 50 individuals (MMFF-STR-EUCL-500-50).

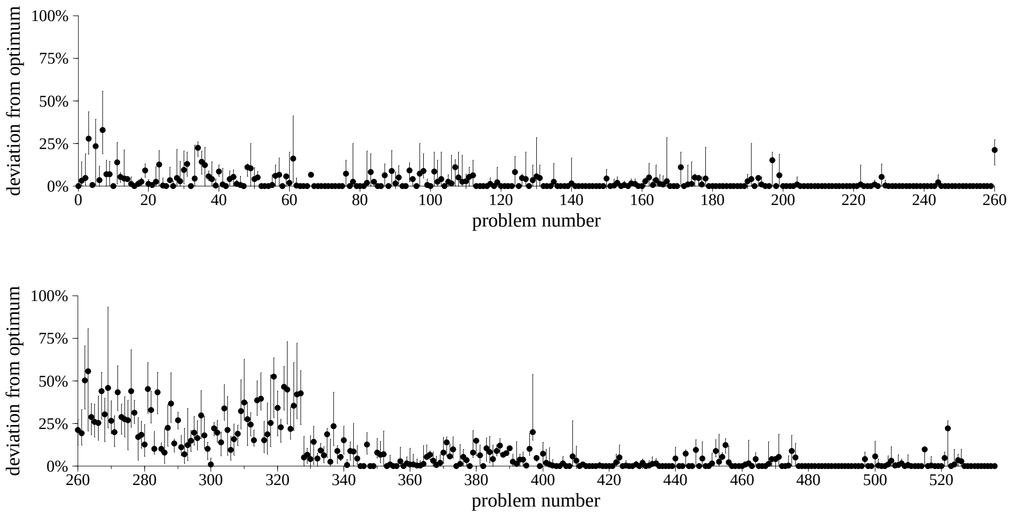


Figure 27 Summary of best, mean, and worst genetic algorithm performance on the multi-mode full factorial problem set using a steady-state genetic algorithm for 500 generations with a population size of 50 individuals (MMFF-STR-SEQ-500-50).

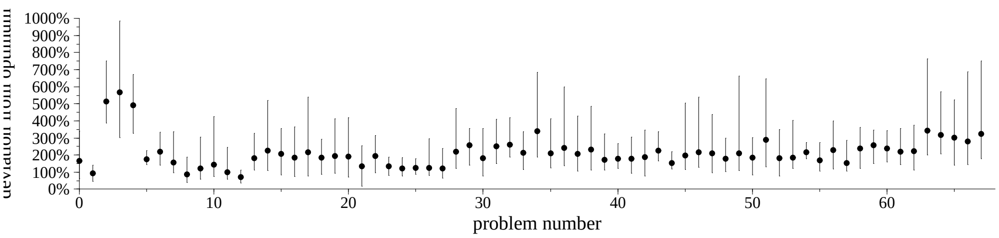


Figure 28 Summary of best, mean, and worst genetic algorithm performance on the jobshop problems using a steady-state genetic algorithm for at most 2000 generations with a population size of 50 individuals (JS-SS-2000-50).

In general, the genetic algorithm took more time than would the equivalent enumerative search or heuristic scheduler. However, it is important to note that no attempt was made to tune the genetic algorithm parameters. This set of tests focused entirely on creating a representation and set of operators for a baseline comparison; these results represent the worst-case for this algorithm and representation.

One important area in which the genetic algorithm out-performed the exact solution method of Sprecher *et al* was the multi-modal problems. The genetic algorithm performed well on some problems that were very difficult for the branch and bound techniques (i.e. the branch and bound method took a long time to find the optimal solution).

Typical run times for a single evolution ranged from a few seconds for 100 generations on a small Patterson problem to over one hour for 5,000 generations on a large jobshop problem.

5.3 Implementation details

All of the tests were run using a single implementation of the genetic algorithm; although minor changes were made to read various data formats and to accommodate different sets of objectives and types constraints, no changes to the genome or genetic algorithm were required.

The implementation was written in C++ using GAlib, a C++ library of genetic algorithm components developed by the author. Tests were run on a variety of Silicon Graphics workstations with MIPS R4x00 CPUs running at 100 to 150 MHz.

6. Conclusions

There is a distinct need for more realistic problem sets. In particular, no problem sets exist with multiple objectives, and the few that include multiple execution modes are far too easy. Only the Benchmarx set includes temporal constraints. Creating such problem sets is no trivial matter; these problems are difficult to formulate even when many simplifying assumptions are made. The Benchmarx set is a step in the right direction.

The genetic algorithm performed best (compared to exact solution methods) on the problems with multi-modal activities. The extra combinations introduced by the multiple execution modes did not hurt the genetic algorithm performance. In fact, in some cases it made the problem easier for the genetic algorithm whereas it made the search more difficult for the branch and bound methods. This suggests that the genetic algorithm (or a hybrid which includes some kind of genetic algorithm variant) is well-suited to more-complicated problems with a mix of continuous and discrete components.

As illustrated in Figures 23 and 24, the genetic algorithm did not perform well on problems in which the resources were tightly constrained. This comes as little surprise since the representation forces the genetic algorithm to search for resource-feasibility, and tightly constrained resources mean fewer resource-feasible solutions. As is the case with most optimization methods, adding more constraints correlates to increased difficulty in solving the problem.

As illustrated in Figure 28, the genetic algorithm did not perform well on the jobshop problems. This is due to the structure of the jobshop problems. As illustrated in Figure 3, the jobshop problems are typically parallel in nature. Since the representation uses relative times, modification of a single value affects all successive activities if they depend strictly upon the predecessor tree of the activity being modified. As a result, one small change has a great affect on a large part of the schedule. A typical project plan, on the other hand, has more interconnections, so a change to a single activity may not affect directly as many successors.

The struggle genetic algorithm consistently found better solutions than the steady-state algorithm at some cost in execution time. Since it must make comparisons and often discards newly created individuals, the struggle genetic algorithm performs more evaluations than the steady-state genetic algorithm, but it *always* found feasible solutions, whereas in some runs the steady-state algorithm did not. The struggle algorithm deserves more study, in particular with respect to comparison methods of genomes and parallelization of the algorithm.

The representation described in this work is minimal (or nearly so) for this class of problems. If, as Davis notes [Davis 85], there is an inverse relationship between knowledge in a representation and its performance, then the methods described in this work can be improved upon a great deal.

What can be done to improve the genetic algorithm performance? Hybridize the representation and/or algorithm and improve the operators. Combining the genetic algorithm with another search algorithm should provide immediate improvement. A hybrid representation that explicitly contains both the resource-constraints as well as the precedence constraints would permit the algorithm to attack the problem from both the resource-constraint perspective as well as the precedence/temporal constraint perspective. Alternatively, a hybrid that maintains both absolute and relative times but operates on one or the other depending on the problem complexity and/or structure might improve the poor performance on problems with parallel structure such as the job shop problems. Finally, the crossover and mutation operators can be tuned to adapt to specific problem structures. For example, one might use a mutator that looks at the parallel/serial nature of the precedence relations as it makes its modifications.

Initializing to good solutions should improve performance. During the course of this work, a number of different initialization approaches were implemented, including initialize-to-critical-path-resource-violated, initialize-to-resource-feasible, and initialize-to-random-delay-times. With the representation and operators described in this work, initializing to a CPM-feasible or resource-feasible set did not help much. This is due in part to the simplicity of the problems; in most cases a feasible solution could be found simply by extending the delay times between tasks. Since there is no direct correlation between a resource-feasible and resource-infeasible solution, in some cases the search space is so sparse that the process of initializing to a set of resource-feasible solutions results in a population of identical (or genetically similar) solutions.

Genetic algorithms are conceptually simple and well-suited to problems with a mix of continuous and discrete variables. However their implementation is far from trivial. Although the basic ideas are straightforward, there is actually a great deal of work (at this point it is still an art) to implementing genetic algorithms on real problems with large search spaces.

7. References

7.1 Sources

This document and the test problems described in this document are available from ftp://lancet.mit.edu/pub/mbwall/phd/

GAlib is available from http://lancet.mit.edu/ga/

The job-shop problems and many other operations research problems are available from http://mscmga.ms.ic.ac.uk/info.html

The *ProGen* program and PSPLIB problems (the SMCP, SMFF, MMFF problem sets) are available from ftp://ftp.bwl.uni-kiel.de/pub/operations-research/

The benchmarx problem is located at http://www.neosoft.com/~benchmrx/

7.2 Bibliography

Aarts, E., P. Laarhoven, et al. (1988). "Job Shop Scheduling by Simulated Annealing." Technical Report OS-R8809 Centre for matematics and computer science(Amsterdam).

Adams, J., E. Balas and D. Zawack (1988). "The shifting bottleneck procedure for job shop scheduling". Management Science 34, 391-401.

Applegate, D., and W. Cook (1991). "A computational study of the job-shop scheduling instance", ORSA Journal on Computing 3, 149-156.

Bagchi, S., S. Uckum, et al. (1991). "Exploring Problem-Specific Recombination Operators for Job shop Scheduling." Proceedings of the Fifth International Conference on Genetic Algorithms.

Balas, E., J. Adams, et al. (1988). "The Shifting Bottleneck Procedure for Job-Shop Scheduling." Management Science 34(3): 391-401.

Blazewicz, J., J. K. Lenstra, et al. (1983). "Scheduling Subject to Resource Constraints: Classification and Complexity." Discrete Applied Mathematics 5: 11-24.

Boctor, F. F. (1990). "Some Efficient Multi-Heuristic Procedures for Resource-Constrained Project Scheduling." European Journal of Operational Research 49(1): 3-13.

Boctor, F. F. (1993). "Heuristics for scheduling projects with resource restrictions and several resource-duration modes." International Journal of Production Research 31(11): 2547.

Boctor, F. F. (1994). "A new and efficient heuristic for scheduling projects with resource restrictions and multiple execution modes." Document de Travail 94-46. Québec, Canada, Groupe de Recherché en Gestion de la Logistique.

Boctor, F. F. (1994). "An adaptation of the simulated annealing algorithm for solving resource-constrained project scheduling problems". Document de Travail 94-48. Québec, Canada, Groupe de Recherche en Gestion de la Logistique.

Bruns, R. (1993). "Direct Chromosome Representation and Advanced Genetic Operators for Production Scheduling." International Conference on Genetic Algorithms: 352-359.

Cleveland, G. A. and S. F. Smith (1989). "Using Genetic Algorithms to Schedule Flow Shop Releases." Proceedings of the Third International Conference on Genetic Algorithms.

Davis, L. (1985) “Job Shop Scheduling with Genetic Algorithms.” Proceedings of an International Conference on Genetic Algorithms and their Applications, Pittsburgh, Lawrence Erlbaum Associates.

Davis, E. W. and G. E. Heidorn (1971). “An Algorithm for Optimal Project Scheduling under Multiple Resource Constraints.” Management Science 17(12): B-803-b817.

Davis, E. W. and J. H. Patterson (1975). “A Comparison of Heuristic and Optimum Solutions in Resource-Constrained Project Scheduling.” Management Science 21(8): 944-955.

Demeulemeester, E. and W. Herroelen (1992). “A Branch-and-Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem.” Management Science 38(12): 1803.

DeJong, Kenneth A. (1975). “An analysis of the behavior of a class of genetic adaptive systems.” Dissertation Abstracts International 36(10), 5140B; UMI 76-9381. University of Michigan, Ann Arbor.

Eshelman, L. J. and J. D. Schaffer. (1992). “Real-Coded Genetic Algorithms and Interval-Schemata”. In L Darrel Whitley (ed), Foundations of Genetic Algorithms 2. San Mateo, CA, Morgan Kaufmann Publishers.

Fisher, H., and G.L. Thompson (1963). “Probabilistic learning combinations of local job-shop scheduling rules”. J.F. Muth, G.L. Thompson (eds.), Industrial Scheduling, Prentice Hall, Englewood Cliffs, New Jersey, 225-251.

Fox, B. and M. Ringer, (1995). “The BENCHMRX Problems” http://www.neosoft.com/~benchmrx/

Fox, M. S., and S. F. Smith (1984). “ISIS - a knowledge-based system for facotry scheduling.” *Expert Systems*, 1(1):25-49.

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

Goldberg, D. E. and J. Richardson (1987). “Genetic Algorithms with sharing for multimodal function optimization”. Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ.

Grüninger, Thomas (1996). “Multimodal Optimization using Genetic Algorithms.” Master Thesis, Stuttgart University.

Harvey, W. D., and M. L. Ginsberg (1995). “Limited Discrepancy Search.” CIRL, University of Oregon, Eugene, OR, USA.

Held, M. and R. M. Karp (1962). “A Dynamic Programming Approach to Sequencing Problems.” Journal of the Society for Industrial and Applied Mathematics 10(1): 196-210.

Hildum, D. (1994). “Flexibility in a Knowledge-based System for Solving Dynamic Resource-Constrained Scheduling Problems”. Umass CMPSCI Technical Report 94-77, University of Massachusetts, Amherst.

Hilliard, M. R., G. E. Liepins, et al. (1988). “Machine Learning Applications to Job Shop Scheduling.” Proceedings of the AAAI-SIGMAN Workshop on Production Planning and Scheduling.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press.

Husbands, P. and F. Mill (1991). “Simulated Co-Evolution as the Mechanism for Emergent Planning and Scheduling.” Proceeding from the International Conference on Genetic Algorithms and their Applications, 264-270.

Husbands, P. (1996). “Genetic Algorithms for Scheduling.” AISB Quarterly, No 89.

Johnson, T.J.R (1967). “An algorithm for the resource-constrained project scheduling problem.” Doctoral Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

Kanet, J. J. and V. Sridharan (1991). “PROGENITOR: A Genetic Algorithm for Production Scheduling.” Wirtschaftsinformatik.

Kirkpatrick, S., C. D. Gelatt, et al. (1983). “Optimization by Simulated Annealing.” Science 220: 671-680.

Kolisch, R., A. Sprecher, and A. Drexl (1992). “Characterization and Generation of a General Class of Resource-Constrained Project Scheduling Problems”. Institut für Betriebswirtschaftslehre, Universität zu Kiel.

Kolisch, R. (1995). *Project Scheduling under Resource Constraints*. Heidelberg, Physica-Verlag.

Lawler, E. L. and D. E. Wood (1966). “Branch and Bound Methods: A Survey.” Operations Research 14(4): 699-719.

Lawrence, S. (1984). “Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Supplement)”, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Lawrence, S. R. and T. E. Morton (1993). “Resource-Constrained Multi-Project Scheduling with Tardy Costs: Comparing Myopic, Bottleneck, and Resource Pricing Heuristics.” European Journal of Operational Research 64(2): 168-187.

Mahfoud, Samir W. (1995). “Niching Methods for Genetic Algorithms.” University of Illinois at Urbana-Champaign, IlliGAL Report 95001.

Mori, M., and C. C. Tseng (1996). “A Genetic Algorithm for Multi-mode Resource Constrained Project Scheduling Problem.” European Journal of Operational Research (to appear).

Michalewicz, Z. (1994). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Berlin.

Müller-Merbach, H. (1967). “Ein Verfahren zur Planung des optimalen Betriebsmitteleinsatzes bei der Terminierung von Großprojeckten” Zeitschrift für wirtschaftliche Fertigung, Vol. 62, pp. 83-88, 135-140.

Muth, J. F., and G. L. Thompson (eds) (1963). *Industrial Scheduling*. Prentice-Hall, Englewood Cliffs, NJ.

Nakano, R. (1991). “Conventional Genetic Algorithm for Job Shop Scheduling.” Fifth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers.

NASA (1962). *PERT/Cost System Design, DOD and NASA Guide*. Washington DC, Office of the Secretary of Defense, National Aeronautics and Space Administration.

Neumann, K. (1990). *Stochastic Project Networks - Temporal Analysis, Scheduling, and Cost Minimization*. Berlin, Springer-Verlag.

Palmer, G. (1994). “An Integrated Approach to Manufacturing Planning”. University of Huddersfield.

Panwalkar, S. S. and W. Iskander (1977). “A Survey of Scheduling Rules.” Operations Research 25(1): 45-61.

Patterson, J. H. (1984). “A Comparison of Exact Approaches for Solving the Multiple Constrained Resource, Project Scheduling Problem.” Management Science 30(7): 854.

Rechenberg, Ingo (1973). *Evolutionsstrategie*. Fromman-Hozboog Verlag.

Sadeh, N. (1991). “Look-Ahead Techniques for Micro-Opportunistic Job Shop Scheduling.” PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1991.

Sampson, S. E. and E. N. Weiss (1993). “Local Search Techniques for the Generalized Resource Constrained Project Scheduling Problem.” Naval Research Logistics 40(5): 665.

Slowinski, R. and J. Weglarz, Eds. (1989). *Advances in Project Scheduling*. Amsterdam, Elsevier.

Smith, S. F., and P. Ow (1985). “The use of multiple problem decompositions in time constrained planning tasks”. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI), vol. 2 pages 1013-1015, Los Angeles, CA.

Sprecher, A. (1994). “Resource-Constrained Project Scheduling: exact methods for the multi-mode case.” Lecture Notes in Economics and Mathematical Systems 409.

Sprecher, A. and A. Drexl (1996). “Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel”. Kiel, Universität Kiel.

Stinson, J. P., E. W. Davis, et al. (1978). “Multiple Resource-Constrained Scheduling Using Branch and Bound.” AIIE Transactions 10(3): 252-259.

Storer, R.H., S.D. Wu, and R. Vaccari (1992). “New search spaces for sequencing instances with application to job shop scheduling”, Management Science 38, 1495-1509.

Syswerda, G. (1990). “The Application of Genetic Algorithms to Resource Scheduling.” Proceedings from the Fourth International Conference on Genetic Algorithms : 502-508.

Syswerda, G. (1991). “Schedule Optimization Using Genetic Algorithms.” Chapter 21 of the *Handbook of Genetic Algorithms*, New York, New York, Van Nostrand Reinhold.

Tavares, L. V. and J. Weglarz (1990). “Project Management and Scheduling: A Permanent Challenge for OR.” European Journal of Operational Research 49(1-2).

Tseng, C. C., and M. Mori (1996). “A Genetic Algorithm for Multi-mode Resource-Constrained Multi-Project Scheduling Problems.” Department of Industrial Engineering and Managent, Tokyyo Institute of Technology, Tokyo, Japan.

Wallace, D. (1994). “A Probabilistic Specification-based Design Model: applications to design search and environmental computer-aided design”. Doctoral Thesis, Mechanical Engineering. Massachusetts Institute of Technology, Cambridge,.MA, USA

Whitley, D., T. Starkweather, et al. (1989). “Scheduling and the Travelling Salesmen: The Genetic Edge Recombination Operator”. Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers.

Yamada, T., and R. Nakano (1992), “A genetic algorithm applicable to large-scale job-shop instances”, Manner, R., and B. Manderick (eds.), *Parallel instance solving from nature 2*, North-Holland, Amsterdam, 281-290.

8. Appendix A - Glossary

allele	One of a set of possible values for a gene. In a binary string genome, the alleles are 0 and 1.
chromosome	A set of information that encodes some of an individual's traits. In evolutionary algorithms, chromosome is often used to refer to a genome.
crossover	A genetic operator that generates new individuals based upon combination and possibly permutation of the genetic material of ancestors. Typically used to create one or two offspring from two parents (sexual crossover) or a single child from a single parent (asexual crossover).
crowding	A niching method in which speciation is encouraged by replacing individuals in the current population with newly generated individuals that share the same characteristics.
deme	A population of individuals. Members of a deme typically share common traits.
due date	The time at which a job is supposed to be finished.
earliest finish	The earliest time at which an activity can be completed.
earliness	The amount of time between the due date and actual finish time for an activity.
evolutionary algorithm	A class of stochastic algorithms based on simplifications of natural evolutionary processes such as selection, survival-of-the-fittest, mating, mutation, and extinction.
exploitation	Local search.
exploration	Global search.
gene	The smallest unit in a genome. In a binary string genome, the bits are genes. In an array of characters, each character in the array is a gene.
genetic algorithm	An evolutionary algorithm in which a population of individuals is evolved using selection, crossover, and mutation. Originally devised as a model of evolutionary principles found in Nature, genetic algorithms have evolved into a stochastic, heuristic search method. A genetic algorithm may operate on any data type with operators specific to the data type.
genetic programming	The use of genetic algorithms to evolve programs. Genetic programming typically uses tree genomes (or tree genomes in combination with other data structures) to represent parse trees. A genetic algorithm then evolves trees using the parsed tree's performance as the objective function.
genome	A complete representation of the information required to characterize the traits of an individual. In evolutionary algorithms, a single solution to a problem.
genotype	The genetic traits of an individual. In a binary-to-decimal genome, the bits are the genotype.
idle time	The amount of time a resource is not actually working on an activity.
latest finish	The latest time at which an activity can be completed.

makespan	The amount of time required to complete a set of activities.
migration	The transfer of individuals from one population to another population.
mutation	A genetic operator that modifies the genetic material of an individual.
non-renewable resource	Also referred to as consumable resources. Non-renewable resources, once used, are no longer available. Examples of non-renewable resources include money and raw materials.
phenotype	The physical traits of an individual. In a binary-to-decimal genome, the decimal values are the phenotypes.
renewable resource	Also referred to as non-consumable resources. Renewable resources are refreshed each period. The length of the period is undefined. Examples of renewable resources include labor and computer time.
sharing	A niching method in which the fitness of similar individuals is deprecated in order to encourage speciation.
slack time (float)	The amount of time an activity can be delayed before further delays will delay any successor activities.
speciation	Also referred to as *niching*, *speciation* is the development of a group of individuals who share the same genetic composition (a species) within a population.
tardiness	The amount of time between the due date and actual finish time for an activity.