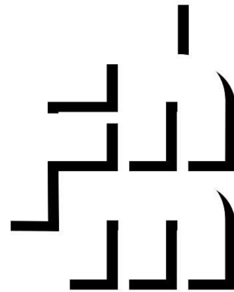
Modeling Resource Alternatives in Project Scheduling

Robert Lemmen <robertle@semistable.com>

29th March 2005



Munich University of Applied Sciences

Supervisor: Prof. Dr. Klaus Köhler
2nd Supervisor: Prof. Dr. Wolfgang Streng

Abstract

Genetic algorithms use an evolutionary approach to find reasonably good results in reasonable short time for hard problems that would otherwise be computationally very expensive to solve. This property makes them desirable to deal with the "resource constrained project scheduling problem", for which one has to find start times for a set of jobs so that some defined constraints are not violated and the last job is finished as early as possible. These constraints are precedence relations between the jobs and resources which are used by the jobs to a varying degree and which have a limited capacity. To allow the representation of flexible project schedules in which there are different possible ways to achieve the desired result, an extension called "multiple modes" has been proposed in the literature. In this thesis we show limitations of this approach and propose an additional extension of the problem representation and the algorithm that helps to alleviate these shortcomings: the resource alternative. Theoretical observations and practical tests show that this extension allows us to solve all traditional problems without drawbacks, but also to deal with problems that were impossible or computationally very expensive to solve with traditional methods.

Erklärung

gemäß § 31 Abs. 7 RaPO

Hiermit erkläre ich, dass ich die Diplomarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

____________________	____________________
Familienname, Vorname	Ort, Datum
Geburtsdatum	Studiengruppe
	Unterschrift

Contents

1 Introduction **6**

1.1 The Project Scheduling Problem 6

1.1.1 Definition of Resource Constrained Project Scheduling 7

1.1.2 Complexity 8

1.1.3 Methods 9

1.2 Genetic Algorithms 12

1.2.1 Problem-specific Definitions 13

1.2.2 Problem-independent Implementation 15

1.2.3 Configuration 15

1.3 Project Scheduling with Genetic Algorithms 16

1.3.1 Problem Representation 16

1.3.2 Genome 17

1.3.3 Operators 18

1.3.4 Fitness Function 19

1.3.5 Results 19

1.4 The Multi-Mode Extension 19

1.4.1 Motivation 19

1.4.2 Definition of Multiple Modes 20

1.4.3 Implementation 21

1.4.4 Results 24

1.4.5 Shortcomings 24

2 Resource Alternatives **26**

2.1 Introduction 26

2.1.1 Motivation 26

2.1.2 Definition 26

2.2 Implementation 28

2.2.1 Problem Representation 28

2.2.2 Genome 29

2.2.3 Operators 30
2.2.4 Fitness Function 31
2.2.5 Coding 32
2.2.6 Parametrization 34

3 Results **42**
3.1 Testing Method 42
3.1.1 Test Cases 42
3.1.2 Transformations 42
3.1.3 Test Setup 45
3.2 Empirical Results 45
3.3 Time and Space Complexity 47

4 Discussion **49**
4.1 Achieved Goals 49
4.2 Conclusion and Future Work 50

List of Figures **52**

References **53**

1 Introduction

Since this work deals with the extension of an algorithm to solve a defined problem class, it is necessary to know the basic algorithm, the problem we are trying to solve and the traditional extensions we are building on. Since both the project scheduling problem and genetic algorithms are fairly complex, this introduction will take a bit longer than usual. Readers that are familiar with the project scheduling problem and the way they are solved with genetic algorithms may skip some of the introductory sections.

1.1 The Project Scheduling Problem

Projects in the way we use the word here are larger undertakings that can be broken into a number of smaller jobs that have to be carried out in order to finish the project. Some of these jobs require that some other jobs are already finished (you wouldn't want to paint a wall that hasn't been built yet), and some of them require the availability of some resources (paint and brushes for example). The main task of project management, and the algorithms we will discuss in this work, is to schedule these jobs in an optimal manner. An optimal project schedule could be one where the whole project is finished as early as possible. Other properties that need to be optimized are also possible, but not considered in this work. One could for example try to optimize the usage of non-renewable resources (a budget), or towards combined goals like the lowest possible budget without violating a dead line.

Projects like this can be found in various areas, typical examples include construction or engineering tasks, but also organizing bigger events or research and development tasks. In fact, good project management is seen as one of the key factors that contribute to the success of larger endeavors. And besides forecasting and controlling the time needed for a job, the project scheduling problem is a central problem of project management, so it is very interesting for many projects.

Unfortunately project scheduling is a very difficult task, and proper methods to deal with it have not been developed and researched earlier than the 1950's and led to the critical path method [14]. At that time limited resources were not considered, which made the whole problem somewhat easier, but also less applicable to real-world projects. Resources represent limited goods that are needed in order to finish a job. They are either renewed after the job is finished, or stay at the decreased capacity. Renewable resources can be used to model machinery or working force, nonrenewable material and money.

Nonrenewable resources are less interesting in the scope of this work, and also lead to less problems when scheduling projects, so they will be ignored for now.

In the following sections we will formalize the resource constrained project scheduling problem, show that it is hard to solve and give an overview of possible ways to approach it.

1.1.1 Definition of Resource Constrained Project Scheduling

As stated before, each project consists of a number of smaller jobs. We will call the number of jobs J and refer to the individual job as j_i with i ranging from 1 to J. The set of jobs that make up the project is then $\mathcal{J} = \{j_1, \ldots, j_J\}$. The duration of a job j is denoted by p_j. We use abstract terms of duration and time here. This could easily mean "working days" or something similar and probably needs to be mapped to real-world time with an appropriate function. We also use discrete multiples of one period and do not preempt jobs once they are started.

Additionally we have precedence relations between jobs in the set. So we have a set $\mathcal{P}_j$ for every job j indicating that every job $i \in \mathcal{P}_j$ has to be finished before j can be started. These precedence relations can easily be expressed as a directed graph over the jobs that has to be acyclic.

We also have a set of K resources $\mathcal{K} = \{k_1, \ldots, k_K\}$ with a capacity of r_k each. As stated before this capacity is renewed after the job that used it up is stopped. Each job j requires r_{kj} units of the resource k while it is active.

We use two additional activities j_0 and j_{J+1} with $p_0 = p_{J+1} = 0$ and $r_{k0} = r_{kJ+1} = 0$ for all $k \in \mathcal{K}$. j_0 is in $\mathcal{P}_j$ for every j, and all j_i for $i \in 0 \ldots J$ are in $\mathcal{P}_{J+1}$. We call these jobs "dummy" jobs as they are not real jobs of the project, don't take up time and don't use up resources. They are only used to simplify things: since no job can be started before j_0 we have a defined start job, and since every job has to be finished before j_{J+1} we always have the same finishing job. These two jobs are useful for the implementation of the algorithm, but have no impact on theoretical observations and can safely be ignored in the scope of this work.

For the problem above we are looking for a solution, that is a start time s_i for each job j_i. As we will see in section 1.1.3 we can derive these start times from a so-called activity list, that is a permutation of the jobs in $\mathcal{J}$ for which all precedence relations are fulfilled.

Let's assume a very simple project as an example: we only have a single resource with a capacity of 5 units and 5 jobs with the durations 2, 3, 1, 2,

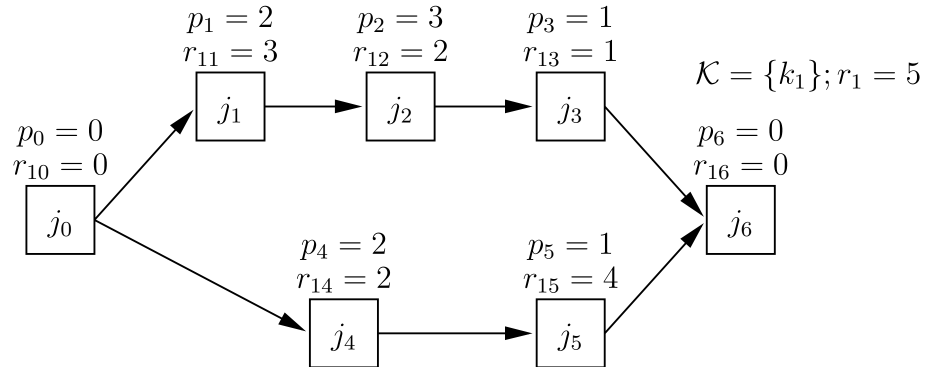


Figure 1: Example project

and 1 as well as resource requests for a single resource with amounts of 3, 2, 1, 2, and 4 respectively. (See figure 1) The algorithms described in this work now have the goal to produce a project schedule as seen in figure 2, where each job now has a start time, no resources are booked in excess of their capacity and the whole project finishes as early as possible. The left part of figure 2 is called "Gantt Chart" and gives the start date and duration for every job in a clear way. The right part represents the usage of the resource by the jobs.

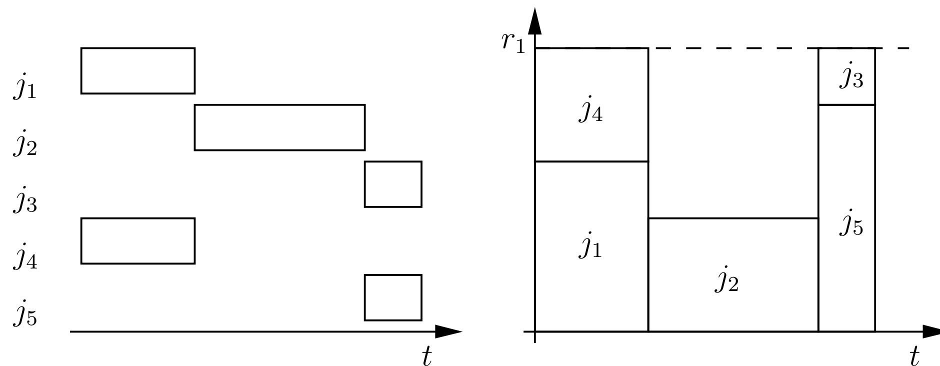


Figure 2: Example project schedule

1.1.2 Complexity

When dealing with optimization problems (like the resource constrained project scheduling problem), it is very important to understand the complexity of the problem and the algorithms we use to solve it. We define the

amount of time $m(n)$ necessary to solve a problem of size n as time complexity or just complexity for short (n being the number of jobs of the project in our case) and then associate the problem with a complexity class. We denote the most important complexity classes with $\mathcal{O}(1)$ (constant), $\mathcal{O}(log(n))$ (logarithmic), $\mathcal{O}(n)$ (linear), $\mathcal{O}(n^k)$ (polynomial) and $\mathcal{O}(k^n)$ (exponential) where k is constant. This classification is important as it tells us whether we can scale our algorithm to larger problem instances or not.

While it is generally fairly easy to figure out the complexity of an algorithm, it is more difficult to determine the complexity of the underlying problem itself. The distinction is important, as we can never find an algorithm to solve a problem that has a lower complexity class than the problem itself. On the other hand we know that there is room for improvements if our algorithm has a higher complexity class than the problem.

A typical way to show the complexity of a problem is to show that another problem of known complexity is in fact a sub-problem and can be expressed in terms of the problem that we are trying to solve (see the works of M.R. Garey and D.S. Johnson[7]). A wide area of known combinatorial problems were shown to be special cases of project scheduling problems: A. Sprecher [22] shows that the job shop scheduling problem, the flow shop problem and the open shop problem are special cases of the resource constrained project scheduling problem, L. Schrage [21] adds the two-dimensional cutting stock problem, Garey et al. [6] the bin packing problem. All of these problems belong to the class of NP-hard problems, which basically means they cannot be solved in polynomial time in a real-world computing situation.

So the baseline for us is that we cannot build an algorithm that finds optimal schedules for our problem in polynomial time, which is very bad as it also means that we cannot find optimal solutions for larger problems. In fact, proven optimal solutions have only been computed for very small problem sizes with about 30 jobs (see the results for the standard problem instances introduced by R. Kolisch et al [17]). When comparing this with real-world projects with many hundred jobs it is obvious that this path doesn't lead us to a usable solution.

1.1.3 Methods

By comparing different methods to solve the project scheduling problem, we can easily differentiate them based on whether they are exact algorithms or heuristics and whether they are deterministic or not.

Exact methods return a provable optimal solution, whereas heuristic methods return a "good" solution that does not necessarily need to be an optimal solution. As we have seen in the previous section, an exact algorithm cannot have a polynomial complexity but must be worse which makes it unusable for larger problems.

Deterministic methods always return the same reproducible result, while non-deterministic algorithms can produce different results for the same problem. In fact, exact methods are mostly deterministic because they always return an optimal solution, but may become non-deterministic should more than one optimal solution exist.

It is important to understand that many of these methods use the fact that you can create a schedule for a sequence of jobs in a deterministic way, and that every possible schedule can be expressed through such a sequence. The construction of the schedule can be done by an algorithm that takes the first job from the sequence of unscheduled jobs and schedules it at the earliest time that it can be started without violating its precedence relations or resource constraints. This is repeated until no jobs are left in the sequence. A sample sequence that would lead to the schedule in figure 2 would be $\{j_1, j_4, j_2, j_5, j_3\}$. Such an algorithm is called "serial generation scheme" or SGS.

An alternative is the "parallel generation scheme" or PGS that iterates over all possible start times in the project in chronological order and always schedules the first job in the sequence that can be scheduled at that time. In this work we only use the SGS, but it has been shown (see [11]) that the PGS and especially an adaptive approach that uses both schemes can also lead to good results.

For exact methods an enumeration scheme to generate all valid sequences (which is less than all possible sequences since precedence relations have to be taken into account) is needed, which is often done by performing a depth-first search on the precedence tree (see e.g. [18]). In order to speed up this approach a number of bounding rules are used, you can e.g. stop searching a sub-tree if the current partial schedule already takes more time than the best known complete one (see [23]). By using smart bounding rules the search space can be cut down significantly, especially if you find "good" solutions early. Heuristics that decide which sub-tree is tried first are often helpful in this regard but don't change the worst-case characteristics of the problem in which you have to check every possible sequence and thus have to deal with a complexity of $\mathcal{O}(n!)$. A lot of work has been done in the field of exact methods for the resource constrained project scheduling problem, and many improvements have been made, but the basic idea is the one outlined above.

In contrast, heuristic methods take the set of jobs, determine which of them are "eligible" and then select one of the eligible jobs and append them to a sequence of jobs. This is repeated until the set is empty and the sequence can be transformed into a schedule as described above. Which jobs are eligible is determined by looking at the precedence graph: if all jobs that have to precede the current one in the schedule are transferred to the sequence, the current job becomes eligible.

The heuristic itself has to choose which of the eligible jobs is selected at a given step in that procedure. Normally this is done by computing a priority for every job and then selecting the job in the set of eligible jobs that has the highest priority. In case of ties some kind of tie-breaking algorithm has to be employed, in the simplest case the job labels are compared. The priority rules used can be classified according to different criteria. We can distinguish between rules that are based on the precedence relations, on the time a job needs to be finished, on the resource constraints, or on combinations of them. Local rules only take the current job into consideration, whereas global rules use many jobs and their properties. Static rules calculate the priority of a job once while dynamic rules have to recalculate it after every step because the current schedule is considered as well. Please note that many, but not all, of these heuristics are completely deterministic. Heuristic methods for project scheduling are usually very fast compared to exact methods and have been scrutinized to a great extent (see [1], [2] and [3]). Unfortunately it is very hard to adapt these heuristics to more complex project representations, and while they perform extraordinarily well on some problems, their average performance is worse than the genetic algorithm used as a basis for this work.

Meta-heuristics are mostly non-deterministic approaches where the basic method is not specific to the problem you want to solve. They generally operate in a standard way on a specific problem representation with specific operators. A good and simple example for this is the simulated annealing method introduced by Kirkpatrick et al. [15]. It takes the physical process of cooling down a melted solid to a solid low-energy state as an example and uses it on mostly arbitrary optimization problems. Starting with a random initial solution a neighbor solution is generated by slightly changing the original solution. If the new solution is better than the old one it is accepted and used as a basis for further search. If it is not better it is only accepted with a probability that depends on the current "temperature". With each step the temperature, and therefore the probability to accept a low-quality solution, is reduced. Simulated annealing can be seen as an extension of the hill-climbing method with the additional possibility to accept solutions of lower quality to escape local optima. Other meta-heuristics are tabu search

(see the basic work by F. Glover [8], [9]), ant systems as proposed by Dorigo et al. [4] and genetic algorithms which we will discuss in detail in the next chapter as they are the foundation of this work.

Project scheduling and related optimization problems are an area of high interest and many sophisticated approaches have been developed that combine the basic solutions outlined above or take them to the edge by applying highly problem-specific heuristics. Nevertheless meta-heuristics in general and genetic algorithms in special, which have not been investigated as long as exact and heuristic methods, have been shown to be a very powerful and flexible approach, especially when using more complicated problem representations as we do in this work.

1.2 Genetic Algorithms

Genetic algorithms are a class of non-deterministic meta-heuristic algorithms that can be employed for many optimization problems and have been used successfully for project scheduling (see the work of S. Hartmann [11]). We use it as a basis for our work and therefore want to explain it in more detail than the other approaches above.

In many cases engineers and scientists have been inspired by nature and the way living organisms solve special problems. Genetic algorithms go one step further and use the way these good solutions are developed in nature itself as an example: the evolution. It would therefore be more precise to talk about evolutionary algorithms instead of genetic algorithms, but the former is often used to refer to a broader set of algorithms in the literature so we will in this work stick to the phrase "genetic algorithms". The usage of genetic algorithms and concepts dates back to the 50ies, but they have not been used and investigated widely until the 70ies. A good introduction can be found in the books by H.J. Holland [13] and D.E. Goldberg [10].

The basic idea behind genetic algorithms is to have a set of solutions out of which you take random existing samples, combine and modify them into new solutions and put them back into the original set. The solutions in the set are assessed according to an optimization criterion and bad solutions are removed from the set while better solutions are kept. Over time the set of solutions as a whole improves and the best solution in it can be picked.

One of the interesting properties of meta-heuristics in general and especially about genetic algorithms is the fact that only a part of the implementation is specific to the problem, while another part is mostly independent of it. We

will therefore describe these two parts independently of each other and only focus on the problem-specific implementation later in this work.

Let us assume we want to find the global maximum of the function in figure 3 so we can refer to this example when talking about the genetic algorithm. Please note that this example is very simple and would in reality be better solved with other approaches. Furthermore a very basic genetic algorithm is described here, many things can and have been done in different ways.

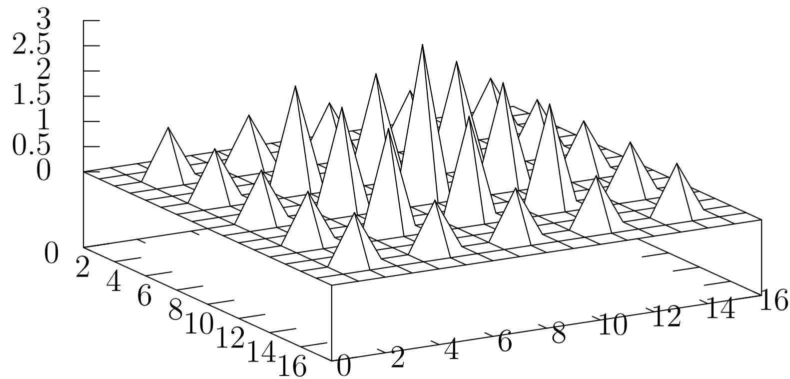


Figure 3: Example problem function

1.2.1 Problem-specific Definitions

A good representation of the solution is often regarded as the main contributor to the performance of an genetic algorithm implementation. In the example above a good and natural representation would be to record an "x" and an "y" parameter that denote the actual position on the problem surface. These parameters will be called "genes". Each gene can have one of a defined set of values that are called "alleles". Genes are grouped into "chromosomes" where each chromosome holds genes of the same type, and the set of chromosomes is called "genome". So in our example we would have a representation that consists of one chromosome with two genes "x" and "y" with alleles in the range of $[0; 16]$ each. In real problems the representation is by far more complex, you will see examples a bit later. A lot has been written about good and bad representations for genetic algorithms, but it

still remains a bit of an art that requires trial and error methods to find good representations for complex problems. It is also important to note that a chromosome needs not necessarily be a list of individual genes, but can have different forms. We will see an example for this later on.

A decoding function is used to translate the representation of the solution into the actual solution. In our example this is trivial as the representation is identical to the solution we were looking for, in other cases this decoding function can be very demanding.

The decoded solution has to be assessed by a fitness function that evaluates to a single value which is used to compare the quality of one solution to another one. In our synthetic example we can simply use the problem function, in reality this can be quite complex especially when multiple independent factors contribute to the actual quality.

A so-called crossover operator then takes two solutions and combines their representation into one or two new solutions. Theoretically more than two solutions could be used as a basis for this operator, but this is hardly ever of benefit to the performance of the algorithm. The input solutions are often referred to as "father" and "mother" and the resulting solutions as "daughter" and "son" or as "child" if only one solution is generated. In our case a possible crossover operator would be to swap one of the two genes between father and mother and call the two resulting solutions daughter and son. Figure 4 illustrates this. Usually some kind of randomness is applied here. For example a representation for a higher order problem of the same kind as our example would require more genes. In this case the crossover operator would have to choose which of the daughter's and son's genes are taken from the mother and which from the father, which would be done randomly.

Figure 4: Example crossover operator

After the generation of new solutions through the crossover operator these solutions are modified by a mutation operator that changes the genome in a random way, but makes sure the genome is still valid. In this simple example we could just modify each gene by a random amount with a given probability, in other cases this is a bit more difficult.

1.2.2 Problem-independent Implementation

Once the crossover and mutation operators and the decoding and fitness function are defined, the problem-independent part of the genetic algorithm has to be implemented. Here two solutions are selected from the population by a selection operator, combined and modified via the problem-specific crossover and mutation operator. Then the resulting solutions are decoded and assessed with the corresponding functions and returned into the population. After that bad solutions are removed, normally by just deleting the worst solutions until a given population size is no longer exceeded. The whole procedure described above is done until a given criteria is reached, most often a given solution quality or a stagnation in the improvement of the solution quality.

There are different models for the main cycle of the algorithm, generational models create N new solutions before these are added back to the population, where continuous models add the new solutions back immediately (and are therefore a special case of a generational model with $N = 1$). Different selection operators exist as well which either take a solution from the population completely randomly (uniform selection operator), or modify the probability to choose a solution based on their fitness. We will use a continuous model with a uniform selection operator in our work.

Of course there are many improvements on this simple genetic algorithm that have been discussed in the associated literature and could be used here too. Among the most interesting should be the island mode, which is very well described in the work of Kohlmorgen et al. [16] and basically consists of many small sub-populations that are improved through the basic genetic algorithm independently, but exchange individuals at a given rate. This leads to a more stable and predictable evolution and to a very simple way to parallelize the evolution over multiple processing units. Improvements like this are of course possible, but are not used in this basic work. It is possible and even recommended to use techniques like this in production-quality code.

1.2.3 Configuration

A very important part of the implementation of any genetic algorithm is a good configuration of different settings. This includes at least the population size and the mutation probability, but might also include other settings in more advanced versions of the algorithm. We will focus on these two setting here. Usually they are determined empirically on a big set of example

problems, but other methods (like deriving them from calculated problem metrics) have been proposed.

It is important to configure the population size properly, as a too small population size will lead to an algorithm that is likely to get stuck in local optima. In the case of a population size of 1 you are effectively implementing a special case of the simulated annealing method. Very large population sizes keep bad solutions around for too long, making the algorithm perform sub-optimal. In the extreme case of an infinitely large population size your algorithm is actually just guessing wildly.

Likewise a too high mutation probability will destroy the good properties of the selected solutions, also leading to wild guessing for a mutation probability of 1. Too low mutation probabilities make it unlikely to generate solutions that are not already present or can be generated with the crossover operator.

1.3 Project Scheduling with Genetic Algorithms

This chapter will introduce one way of doing project scheduling with genetic algorithms. A lot of work has been done in this field and many slightly different approaches have been proposed. The approach we use here is basically the algorithm as described by S. Hartmann (see [11]) since it is one of the best approaches so far, reasonably simple and easy to extend. We skip some specialties like local optimization and optimized initial populations as they are irrelevant in the scope of this work.

1.3.1 Problem Representation

In section 1.1.1 the basic definition of the resource constrained project scheduling problem was shown. If we want to represent a specific problem in that domain, we need to hold all necessary information about this specific problem in an easy to access way. We can follow the definition pretty closely here and end up with the following structure:

```
problem := [job]*
           [resource]*

job := name
       duration
   [predecessor_job_reference]*
   [resource_request]*
```

```
resource_request := resource_reference
                    requested_amount

resource := name
            availability
```

1.3.2 Genome

The best representation for our problem found so far are the "activity list" in which you simply have a list of jobs that are scheduled in that order. Unfortunately this is a bit different from the simple genome structure we have used in the example above. We use a single chromosome for the activity list that has one gene for every job in our problem. In order to make the sequence denoted by this chromosome valid it needs to fulfill two criteria:

1. Every gene has to be in the range $[1..J]$ where J is the number of jobs and has to be unique in the chromosome. This means an allele can only be used once in the chromosome.
2. The precedence relations between the jobs have to be fulfilled for the genes.

It is quite easy to see that we end up with a representation that only allows permutations of its chromosome and has additional constraints that disallow certain representations (the precedence relations). Example chromosomes for the problem in figure 2 would look like the ones in figure 5. The two chromosomes to the left are valid and would in fact decode to the same schedule (the one in figure 2), the two chromosomes to the right are invalid because they violate either of the two criteria above.

Valid, equivalent:	Invalid:
1 4 2 5 3	1 4 4 5 3
4 1 2 3 5	2 1 4 5 3

Figure 5: Example chromosomes

1.3.3 Operators

Our crossover operator has to take the criteria above into account and create only valid genomes. We therefore use a standard one-point crossover for permutation-based genotypes as used by many applications (see [20]). Assuming we have a father chromosome $F = (f_1, \ldots, f_n)$ and a mother chromosome $M = (m_1, \ldots, m_n)$ we create a daughter chromosome $D = (d_1, \ldots, d_n)$ and a son chromosome $S = (s_1, \ldots, s_n)$. First we draw a random integer q with $1 \leq q < n$ and then copy the genes at positions 1...q from the father chromosome to the daughter chromosome. Then we fill up the rest of the chromosome with the genes from the mother chromosome that have not already been copied from the father while obeying their order in the mother chromosome. The son chromosome is constructed in the same way, but with the first part taken from the mother and the rest from the father. S. Hartmann [12] (among many others) shows that this operator never violates the two criteria above. Figure 6 illustrates this operator.

Father 1 4 2 5 3
Mother 1 2 4 3 5
$q = 2$
Crossover
1 4 2 3 5 Daughter
1 2 4 5 3 Son

Figure 6: One-point crossover for permutation-based genotypes

Because this operator has been shown to perform sub-optimal when scheduling problems with many jobs, a two-point variant has been added. It basically works the same way, but we draw two random integers q_1 and q_2 so that $1 \leq q_1 < q_2 < n$. We then take the genes at positions $1 \ldots q_1$ from the father, fill up until position q_2 from the mother as above, and then fill up the rest with the remaining genes from the father (see figure 7).

Father 1 4 2 5 3
Mother 4 2 1 3 5
$q_1 = 1; q_2 = 3$
Crossover
1 4 2 5 3 Daughter
4 1 2 3 5 Son

Figure 7: Two-point crossover for permutation-based genotypes

Of course a mutation function is needed as well. A very basic one is used in this work that for $i = 1 \ldots n - 1$ swaps every gene g_i with its successor g_{i+1} with a probability of $p_{mutation}$ if this does not violate the precedence relations.

1.3.4 Fitness Function

The fitness function used is very simple, since we only need to find the latest finishing time in all jobs. Here the dummy activities come in handy, as we just need to do one look-up of a start time, namely of the final dummy activity, to get our result. If we would use more complex fitness functions then this operation would get more complicated too.

1.3.5 Results

A lot of work has been done in the field of solving the project scheduling problem with genetic algorithms, and it is getting more and more clear that genetic algorithms are currently the most useful approach when looking at the problem from a real-world perspective. First of all one needs to be able to find solutions fairly quickly, so you can work with your schedule interactively, which rules out any exact methods. And you need to be able to modify the optimization goals and other problem properties to suit your individual scheduling problem, Heuristic methods are not adaptable enough for this. Because of these reasons genetic algorithms have been chosen as a basis of this work.

1.4 The Multi-Mode Extension

In the previous sections the basic project scheduling problem was defined and shown how it can be solved with genetic algorithms. In this section we will introduce you to one of the extensions of the original problem, multiple modes, why they were invented, how they are implemented and why there still is a problem with them.

1.4.1 Motivation

The whole art of project management boils down to making the right decisions about how to do a certain project. The algorithm described up to now can help to define a schedule, but it cannot help you with other decisions, like how to do a single job. Unfortunately these decisions influence the scheduling process and therefore cannot be made independently. A simple example: You want to move a pile of gravel on a construction site and can choose whether you do it with one caterpillar in two days, or with two caterpillars in one day. The second option would of course hinder the second caterpillar from doing

something other during that period. In some cases, for example if you have nothing else to do for the second caterpillar, you would want to use both. In some other cases, for example if it does not really matter how long you need to move the gravel, you would want to use only one. In order to incorporate this kind of decisions into the project scheduling problem an extension called "multiple modes", that we will discuss in detail over the next sections, has been proposed and discussed in the literature (see [5]).

1.4.2 Definition of Multiple Modes

In section 1.1.1 we defined the basic project scheduling problem. For the multiple mode extension we remove both the resource usage and the make-span from the jobs and add a set of modes the job can be processed in. These modes have a make-span and a set of resource requests each. So we now have a set of J jobs $\mathcal{J} = \{j_1, \ldots, j_J\}$ each job j having a set of M_j associated modes that we denote by $\mathcal{M}_j = \{m_1, \ldots, m_{M_j}\}$ each. Each of these modes m_i has a duration of p_i and requires r_{ki} units of the resource k while it is active. Now every job is scheduled in one of its modes.

This allows us to define multiple alternative ways to do a single job. Taking the example above we would now have two modes for the job that moves the gravel. One mode would require two caterpillars as resources and would take a duration of only one day, the other mode would only require one caterpillar as a resource but take two days of duration. By integrating this decision into the genetic algorithm used to solve the project scheduling problem we have an extended, more complex, problem. But we are also searching a bigger space for optimal solutions that was inaccessible before and we are therefore likely to find better solutions.

A solution for this extended problem would consist of a permutation of the jobs that obeys the precedence relations as in section 1.1.1 and additionally for each job j_i a number n_i denoting which of the M_i modes is used.

Taking the example from section 1.1.1 and extending it to allow multiple modes, we could modify the jobs j_2 and j_5 so that they have two possible modes each. Additionally to the original mode we could add the possibility to do j_2 over a duration of 2 time units instead of 3 at an increased resource allocation of 4 instead of 2 units. For j_5 a mode that allows us to decrease the resource usage from 4 to 3 by increasing the job make-span from 1 to 2 could be added. Figure 8 illustrates this, each box representing a possible mode now. In figure 9 you can see that the original mode of j_5 is used and the new one for j_2 leaving us with a decreased total project make-span.

A possible solution in this example could be a permutation of the jobs like (j_4, j_1, j_2, j_3, j_5) and the selected modes of the jobs would be $n_1 = 1; n_2 = 2; n_3 = 1; n_4 = 1; n_5 = 1$. Please note that for the jobs n_1, n_3, and n_4 the first mode must always be selected, as they only have one.

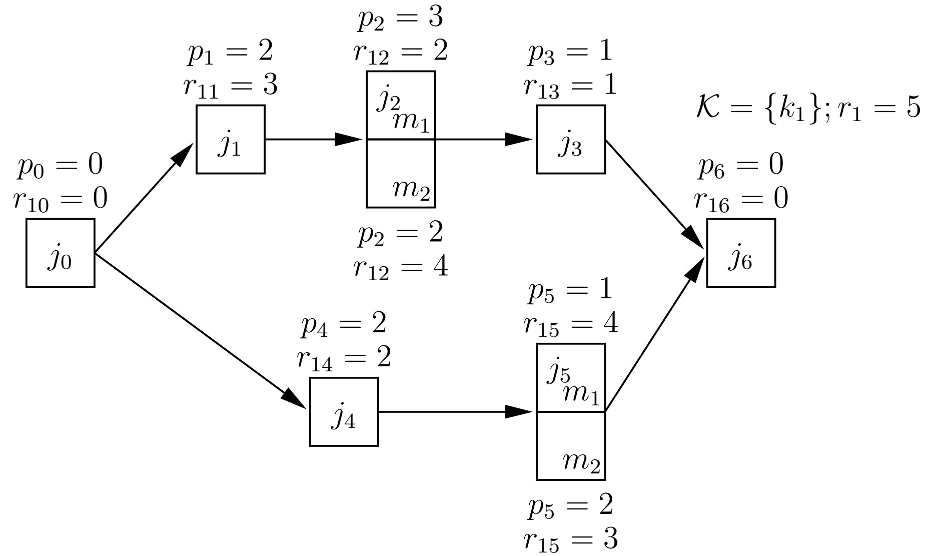


Figure 8: Example multi-mode project

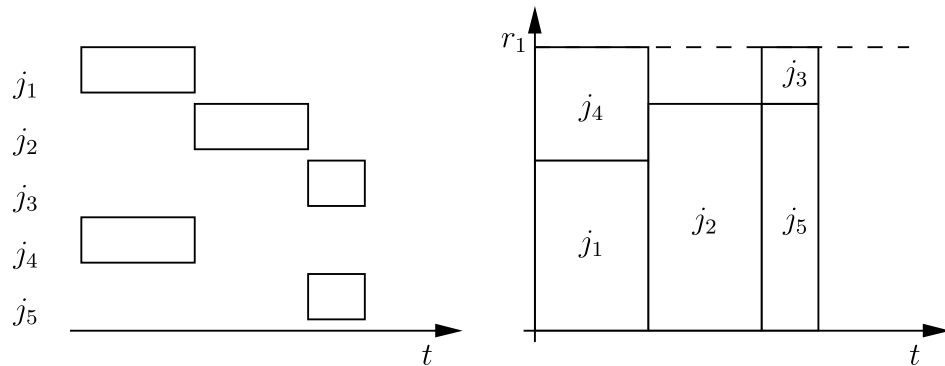


Figure 9: Example multi-mode project schedule

1.4.3 Implementation

In order to implement the multiple mode extension we first need to extend the problem representation from section 1.3.1 according to the new problem definition. It will then look like this:

```
problem := [job]*
           [resource]*

job := name
   [predecessor_job_reference]*
   [mode]*

mode := duration
    [resource_request]*

resource_request := resource_reference
                    requested_amount

resource := name
            availability
```

Please note that while the duration of a job and its resource usage are now located on the mode it is done in, the precedence relations still remain located on the jobs. That means the precedence relations cannot be changed by the multi-mode extension.

Now the solution representation, the genome, needs to be changed in order to reflect the changes to the problem representation. The new genome will have two chromosomes: one with the sequence of jobs as described in section 1.3.2, and one with n genes where the ith gene denotes the mode job j_i is done in. This is illustrated in figure 10, both genomes would be valid solutions to the example in figure 8.

A comparison of the the two chromosomes described reveals that they are very different in their structure: the sequence chromosome consists of genes with alleles from $1 \ldots n$ where each allele can only be used once (a permutation chromosome) whereas the modes chromosome consists of n independent genes that can even have a different number of alleles each, as different jobs can have different numbers of modes. It is very important to understand the the ith gene in the modes chromosome does not define the mode the ith job in the sequence chromosome is done in, but the mode job j_i is done in, no matter what position it might have in the sequence chromosome.

Because of the introduced changes to the the genome, we also have to change the operators that are used to create and modify solutions. The crossover operator for the modes chromosome is very simple: we draw a random number q with $1 \leq q < n$ and use then set the mode genes $1 \ldots q$ in the daughter to have the values of the corresponding genes in the father chromosome and

	Genome #1	Genome #2
Sequence chromosome	1 4 2 5 3	4 1 2 3 5
Modes chromosome	1 2 1 1 1	1 1 1 1 2

Figure 10: Multi-Mode genome

the rest of the genes to the corresponding values in the mother chromosome. Figure 11 illustrates this crossover operator. Of course we can also do a two-point crossover operator similar to the one in section 1.3.2. We draw two random numbers q_1 and q_2 with $1 \leq q_1 < q_2 < n$ and then take the genes at positions $1 \ldots q_1$ from the father, $q_1 + 1 \ldots q_2$ from the mother and $q_2 + 1 \ldots n$ from the father to form the daughter chromosome. Figure 12 illustrates this operator. In both cases the son genome is constructed in the same way, but with swapped parents.

Because of the number of alleles a gene at a given position in the modes chromosome being independent of the sequence chromosome, the values for q_1 and q_2 are independent of the corresponding values used for the sequence crossover. In fact you can even use a one-point crossover for one of the chromosomes and a two-point crossover for the other.

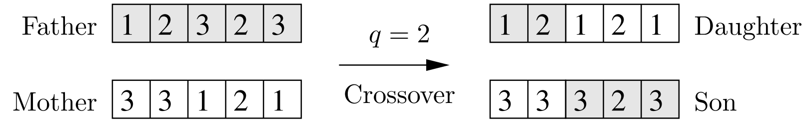


Figure 11: One-point crossover for the modes chromosome

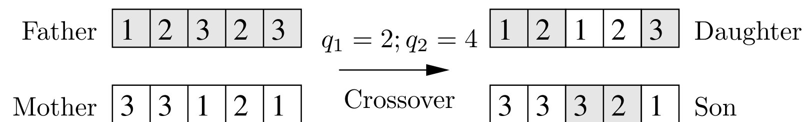


Figure 12: Two-point crossover for the modes chromosome

Of course a new mutation operator is needed as well. We just iterate over the modes chromosome and change every gene to a random, valid value with a defined probability. It is interesting to see that since the different modes for a job have no inherent order, we can just modify the value and don't need to worry about "how much" we modify it.

The modifications of the the decoding and fitness functions are very simple, as we only need to look up the used mode for each job in the modes genome

and use the resource usage and duration of this mode instead of the job's duration and resource usage in the original algorithm.

1.4.4 Results

The multiple mode extension allows the project manager to define alternative ways to do individual jobs, which in turn allows the algorithm to find better solutions since there are simply more options. We have also seen how easy it is to adapt the genetic algorithm to an extended and more complex problem, something that would have been way more difficult for any sufficiently sophisticated exact or heuristic approach. Furthermore, the literature shows that solving the extended problem does not even take much more time than solving the original problem, which gives hope that this also holds for further extensions of the problem.

1.4.5 Shortcomings

Unfortunately the multiple mode extension is not able to cover all problems project managers could face when using project scheduling software based on the original model. To illustrate this, lets look at a simple and realistic example: you have to plan a construction job and have five workers at hand that have different capabilities. They can all shovel around stuff equally well, but two of them can weld and two other workers are allowed to drive a truck. Assuming you have to do three jobs that require one of these capabilities each, you are in trouble: you cannot model the project with the original project scheduling problem representation. To do that you would have to break your work force into resources according to their capabilities, but all of the workers would have to be in the "digging worker" resource, which isn't possible since they are already in "welding worker" or "driving worker" resource. Multiple modes make it possible to schedule this kind of project, as you can simply break up the work force into the individual workers and use modes to represent the different options you have when assigning workers to jobs. This works fine in the simple example above, but breaks down in complex projects because you would need too many modes. Imagine 20 workers that all have an individual set of capabilities and a job for which you need any two of them. You would already need $20 \cdot 19 = 380$ modes to represent this! And now imagine you would also need one of 4 possible, differently rated, welding machines: you are already down with 1520 modes for this single job. This combinatorial explosion of possibilities is the result of two facts: we do not have uniform resource classes but distinct individual

resources, and only a limited way to express alternatives in our model. The first fact is a property of our modern and complex world and the projects we are trying to schedule, and nothing we can change. The second property is more interesting: we are limited in the way a project can be modeled because we have only one degree of freedom for every job: the mode it is scheduled in. We therefore propose an extended model where we can choose a mode for each job, along with the job duration, but have alternatives for every resource request. This adds additional freedom to the modeling of each job and would avoid the combinatorial explosion in modes needed to express a project like the one above.

2 Resource Alternatives

In the previous sections an overview of the project scheduling problem, how to solve it with genetic algorithms and an extension to it: multiple modes was given. In this section an additional extension s proposed and explained why it is beneficial to the project manager. We will refer to this extension as "resource alternatives" from now on.

2.1 Introduction

2.1.1 Motivation

Section 1.4.5 has shown that the multiple modes extensions, while making project modeling more flexible, still makes it very cumbersome to schedule some projects and leads to a combinatorial explosion of the number of modes needed to model such projects. We want to add an additional degree of freedom to the project model in order to make it possible to model such projects in a more natural and compact, that is without the many modes, way.

In order to do so, we replace the resource requests that we had before with a list of alternative resource requests. In the example in figure 13 we have done so for j_4. While there are of course different possibilities to add additional freedom, this way seems very natural and likely to reflect real-world projects (see the examples in section 1.4.5). At the same time it is not as hard to implement and error-prone as other possible solutions.

2.1.2 Definition

Formally defining the project scheduling problem with multiple modes and our new extension is another iterative step in extending the problem, like the the multiple modes extension: we still have a set of J jobs $\mathcal{J} = \{j_1, \ldots, j_J\}$. Each job has a set of M_j modes that we denote as $\mathcal{M}_j = \{m_1, \ldots, m_{M_j}\}$. Each of these modes has a duration of p_{M_j} and a set of R resource requests $\mathcal{R} = \{r_1, \ldots, r_R\}$. Each resource request consists of A resource alternatives $\mathcal{A} = \{a_1, \ldots, a_A\}$ that take up $r_a k$ units of the resource r while the job is active. When scheduling a job the currently active mode has to be determined as well as which resource alternative is being used for every resource request.

A solution to this problem would then consist of the permutation of jobs and a selected mode for each job as before. Additionally we have to select a

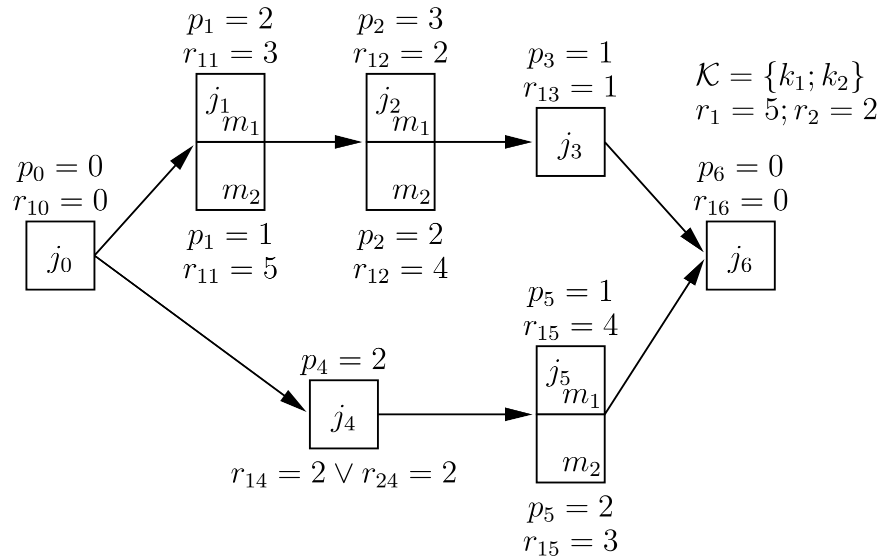


Figure 13: Example project with resource alternatives

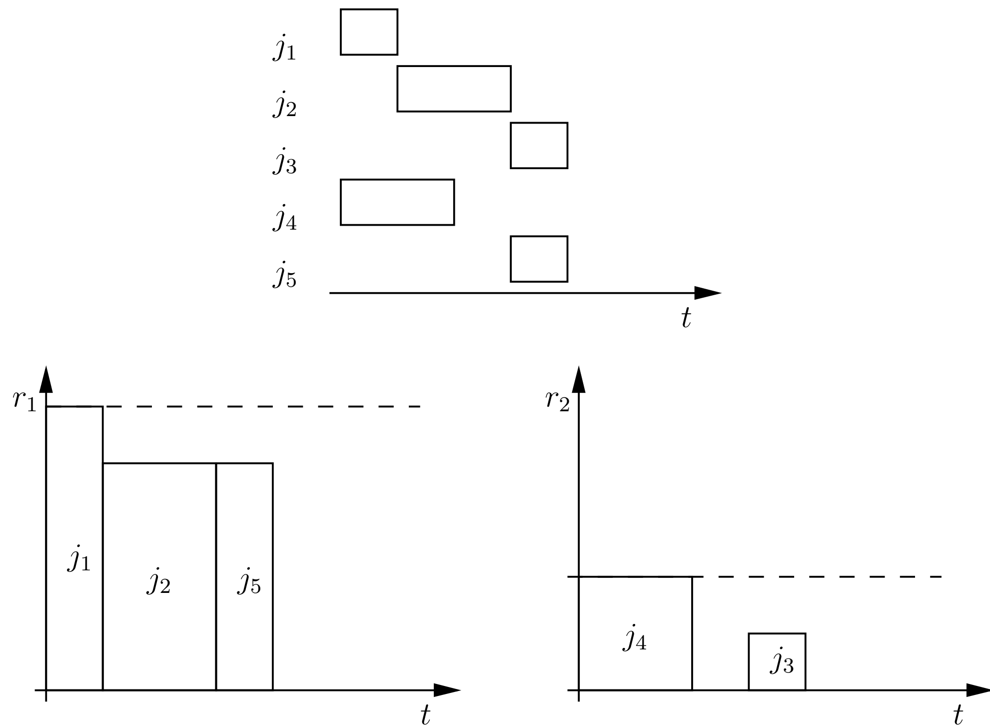


Figure 14: Example project schedule with resource alternatives

resource alternative for every resource request on every mode. In the example in figure 13 there is only one resource request (the only one on j_4) out of the total of 8 resource requests that has more than one alternative, so all other resource alternative selections have to be 1 in this example.

2.2 Implementation

While the formal definition of our extension did not create many problems, the implementation brings up some interesting points. Therefore, and because it is the main part of our work, it is described in detail in this section.

2.2.1 Problem Representation

The extension of the problem representation is quite straightforward as we just follow the extension of the problem definition and add another entity for our resource alternatives to the model:

```
problem := [job]*
           [resource]*

job := name
   [predecessor_job_reference]*
   [mode]*

mode := duration
    [resource_request]*

resource_request := [resource_alternative]*

resource_alternative := resource_reference
                     requested_amount

resource := name
            availability
```

By looking at the code, you will find this structure almost verbatim (some utility fields for internal usage were added) in the declaration file `rcp.h`.

2.2.2 Genome

In order to adapt the genome to the extended problem we need to add a third chromosome that carries information about which resource alternative is used by each resource request. Figure 15 illustrates this. Note that there might be more than one resource request per job, so the alternatives chromosome will be longer than the job sequence chromosome and the modes chromosome. In the left part of the figure this is illustrated by having the different genes on the alternatives chromosome that belong to the same job below each other. In the compressed form, that we use in our code, we simply concatenate these per-job lists of resource alternative genes into one single list. In addition, we also skip genes for modes and alternatives that have only one allele. For example a job that has only one mode does not need a gene that chooses which mode to run in. This makes the coding of the operators easier and the code itself faster.

	Simple representation					Compressed representation							
Sequence chromosome	1	4	2	5	3	4	1	2	3	5			
Modes chromosome	1	2	2	1	3	2	2	1	3				
Alternatives chromosome	2, 1, 4	3	1, 2		4, 3	2	1	4	3	1	2	4	3

Figure 15: Resource alternatives genome

Because this genome is rather complex it might help to have a closer look at how the genes on the different chromosomes are related to each other. In figure 16 you can see a solution to our problem in the compressed form. The sequence chromosome isn't anything new, but the compressed form of the modes chromosome is. As stated before, we leave out all genes for jobs that only have one mode. The lines to the sequence chromosome show for which job this modes gene denotes the mode to use. Again, please note that the ordering of the modes chromosome does not change if the sequence chromosome is modified. This is also true for the alternatives chromosome, which in this case has only one gene because we only have one resource request with more than one alternative. A single job can have many resource requests with multiple alternatives on one or on different modes, so there could be many alternative genes for each job.

An interesting point in this genome is that not necessarily all genes in the alternatives chromosome contribute to the solution, as some of them might belong to a mode that is not being used in an individual solution. These

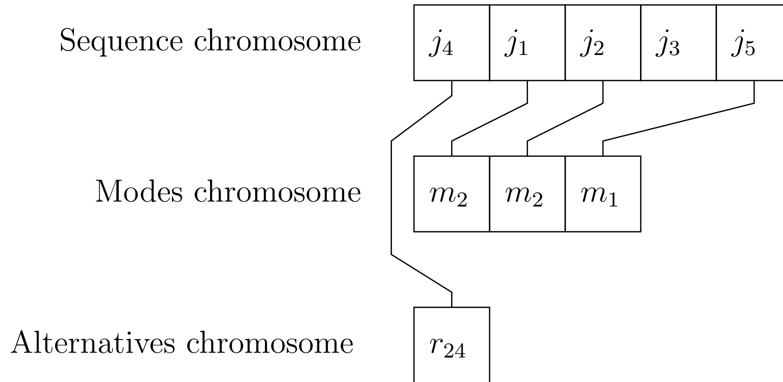


Figure 16: Example chromosome

recessive genes might be a point for further research, especially in regard to whether it proves beneficial to mask them out while applying the mutation operator. Since these genes are not evaluated anyway, any mutation applied to them does not show its effect immediately and might therefore destroy good properties without being noticed. This is a very specific topic that is not looked at in this work as the effects, while being interesting, should be negligible.

In the code our genome is defined in `ga.h` and uses pointers from the problem representations that define which gene corresponds to which mode or alternative. This makes the code for the operators and fitness function much simpler and adds an overhead that is only done once: during the problem initialization.

2.2.3 Operators

Because of the structure of the chromosomes, we can use the same operators for the alternatives chromosome as for the modes chromosome. So we now have one- and two-point as well as mutation operators for the job schedule and for the other two chromosomes, a total of 6 operators. The effort gone into the problem representation and genome clearly pays off here, as can be seen from the really simple implementations of these operators in the file `ops.c`.

One complication does show up however: while the operators in the multi-mode case could not produce invalid solutions, they now can and we have to deal with it. This complication comes from the fact that a resource alternative may together with a resource alternative belonging to another resource

request exceed the availability of the resource. In the traditional problem representation this can never happen as every single resource request must be satisfiable (or the project can not be scheduled), and different requests must not necessarily be scheduled at the same time as they always belong to different jobs. In our new problem representation it is entirely possible that a resource request with a few alternatives is reasonable by itself, no matter what alternative is chosen. But we can as well have two or more requests with multiple alternatives, where for at least one resource the corresponding alternatives sum up to more than the available amount of the resource. In this case, not all combinations of selected alternatives are valid solutions. This kind of problem is not new in the field of genetic algorithms, and there are basically three approaches described in the literature: you could simply drop each invalid solution from the population, you can repair it with a special repair operator, or a combination of these. The first approach is very simple to implement, but has the drawback that you might loose otherwise good solutions because of a single gene being off. Repairing saves the solution, but might be very complicated and require quite some time. Here the mixed approach is used where simple cases are fixed and complicated ones are discarded. This solution is easier to implement, gives us a reasonable upper bound on the time the repair operator might take and still saves most of the solutions that would otherwise be lost. How far you want to go when doing the trade-off between repair operator complexity and the amount of solutions saved is an interesting question that should be researched independently. In our case we simply do a single-pass search over the direct neighborhood of the current solution and stop as soon as we have found a valid genome, and discard the solution if we did not find a valid one. The repair operator is implemented in the file `repair.c`.

2.2.4 Fitness Function

For the decoding function we still use the SGS as described in section 1.1.3 and which is used for the traditional problem as well. Fortunately the algorithm does not get any more complex through the resource alternatives extension as we can do another redirection taking the current alternative and a table with the alternatives. This is the same way that the multiple modes are added to the basic problem. Its an interesting fact that neither the multiple modes nor the resource alternatives add any complexity to the decoding procedure, and we will get back to it later. The fitness function that operates on the decoded schedule is still trivial as we just need to find out the start date of the last (dummy) job.

2.2.5 Coding

In order to test our new extension we need to compare it with the traditional multiple modes approach and therefore need good implementations for both of the algorithms. The resource alternatives extension does not necessarily require the presence of the multiple mode extension, but is more useful with it. If we combine the three approaches, traditional without multiple modes, with them, and with our extension can be implemented in a single program. This is due to the fact that every traditional problem can be expressed as one with multiple modes having only one mode per job. Likewise a multiple modes problem is just a special case of our extension (if you combine it with multiple modes), because its the same as having only one alternative per resource request. In fact, we only need one program to compare the different approaches.

There are a good deal of genetic algorithm frameworks around that one could use when implementing algorithms like those used in this work. These frameworks would take care of the problem-independent part of the genetic algorithms and provide a stable basis to implement the rest. A couple of these frameworks were considered, and two of them selected for a more detailed evaluation: GAUL[1] and GALib[2]. GAUL, while looking pretty nice from the coding has many assumptions about the problems to solve in the problem-independent code, which makes it very hard to configure the algorithm for our problem. GALib seems unsupported and the website is only responsive every now and then, but the code can be found on some FTP mirrors. GALib looks very much like a tool for teaching genetic algorithms and has problems with more complicated problem representations and genomes. For these reasons none of them was chosen but the whole algorithm was implemented from the ground up. If you look at the code now, you will see that this is not such a big problem, as the problem-specific parts, which would have to be written anyway, out-weight the problem-independent code by far if you measure them in lines of code (more than 90% are problem-specific) and are also more complex. So there would not have been much work saved by using such a framework.

The program was implemented in ANSI C for portability and performance. The functionality was grouped into logical units and each of them implemented in an individual `.c` file with an accompanying `.h` file for the declarations. So we have the decoding function in `decode.*`, the fitness function in `fitness.*` the generation of the initial population in `initial.*`, our

[1] http://gaul.sourceforge.net
[2] http://lancet.mit.edu/ga/

crossover and mutation operators in ops.*, the problem representation in rcp.*, the repair operator in repair.*, transformation functions for the three kinds of problems are in transform.* the basic genetic algorithm as well as the genome is defined in ga.*. Besides these files we have some utility functions in lib.h, an input file parser in parser.* and a utility data structure in slist.*. The basic command line parsing and calling the genetic algorithm is done in main.c.

A simple command line program was used which reads its problems from a file because this way it is very easy to automate tests through shell or Perl scripting, we will get to that later. An input file format based on XML was defined that allows to model all kinds of problems that our algorithms are capable of solving. Despite the completeness, the input format definition as an XML DTD is still quite brief:

```
<!ELEMENT problem (resources,jobs)>
<!ATTLIST problem best-known CDATA "0">
<!ATTLIST problem name CDATA "">
<!ELEMENT resources (resource*)>
<!ELEMENT resource EMPTY>
<!ATTLIST resource name CDATA "">
<!ATTLIST resource availability CDATA "0">
<!ELEMENT jobs (job*)>
<!ELEMENT job (successor*,mode*)>
<!ATTLIST job name CDATA "">
<!ELEMENT successor EMPTY>
<!ATTLIST successor name CDATA "">
<!ELEMENT mode (request*)>
<!ATTLIST mode duration CDATA "0">
<!ELEMENT request (alternative*)>
<!ELEMENT alternative EMPTY>
<!ATTLIST alternative resource CDATA "">
<!ATTLIST alternative amount CDATA "0">
```

In order to read these input files our program links against libxml2 from the GNOME project, a free XML library for C, which is available from http://www.xmlsoft.org and should compile fine on most platforms. Apart from that no external dependencies, besides a standard C compiler, are needed. During development gcc 3.3.5 and GNU make were used on a Sun Sparc Ultra 10 running GNU/Linux, but as stated above the program should

compile and run fine on almost any platform. The source code is available from `http://www.semistable.com/thesis-src.tar.gz`, future versions of the code (not necessarily related to this work) will be available from `http://www.librcps.org`.

2.2.6 Parametrization

As outlined in section 1.2.3 it is essential for the performance of any genetic algorithm to do a proper parametrization. In our case we only have a few parameters that need to be selected: the population size and mutation rates for each chromosome. In order to find the best possible parameters for a set of problems one would normally have to run the algorithm for each problem in the set and for each combination of parameters, preferably multiple times because of the probabilistic nature of the algorithm. This is computationally extremely expensive and not feasible in any real-world situation. We therefore use a simpler optimization: we guess a reasonable good value for one of the parameters and optimize another one with this one set to a fixed value. After that we can in turn optimize the guessed one and all the others. This works well only if the contributions of the parameters to the total performance are independent which is hard to verify without searching the whole parameter space, but we can have a look at a part of this space. In figure 17 we can see the performance of our algorithm over a given problem set depending on the population size and schedule mutation probability, without resource alternatives or modes. The bowl-shaped surface suggests that we can use the proposed method, which is also logical as the population size is a problem-independent parameter whereas the mutation probability is highly problem-specific. We will therefore use the method proposed above to get parameters for the population size and schedule mutation rate.

In order to do so we first need a measure of the performance of our algorithm. This is difficult, since we basically want to optimize according to two different criteria: the number of reproductions needed to find a good solution, and the quality of the solution found. Since we know the best possible solution quality for our problems, we can use a very simple measure: we let our algorithm work until either an optimal solution is found or a defined number of reproductions have been performed (in our case 50000). If this number is significantly higher than the number of reproductions typically needed to find a solution we have a measure with the following properties:

1. An optimal solution found within the allowed number of reproductions is better than any other solution.

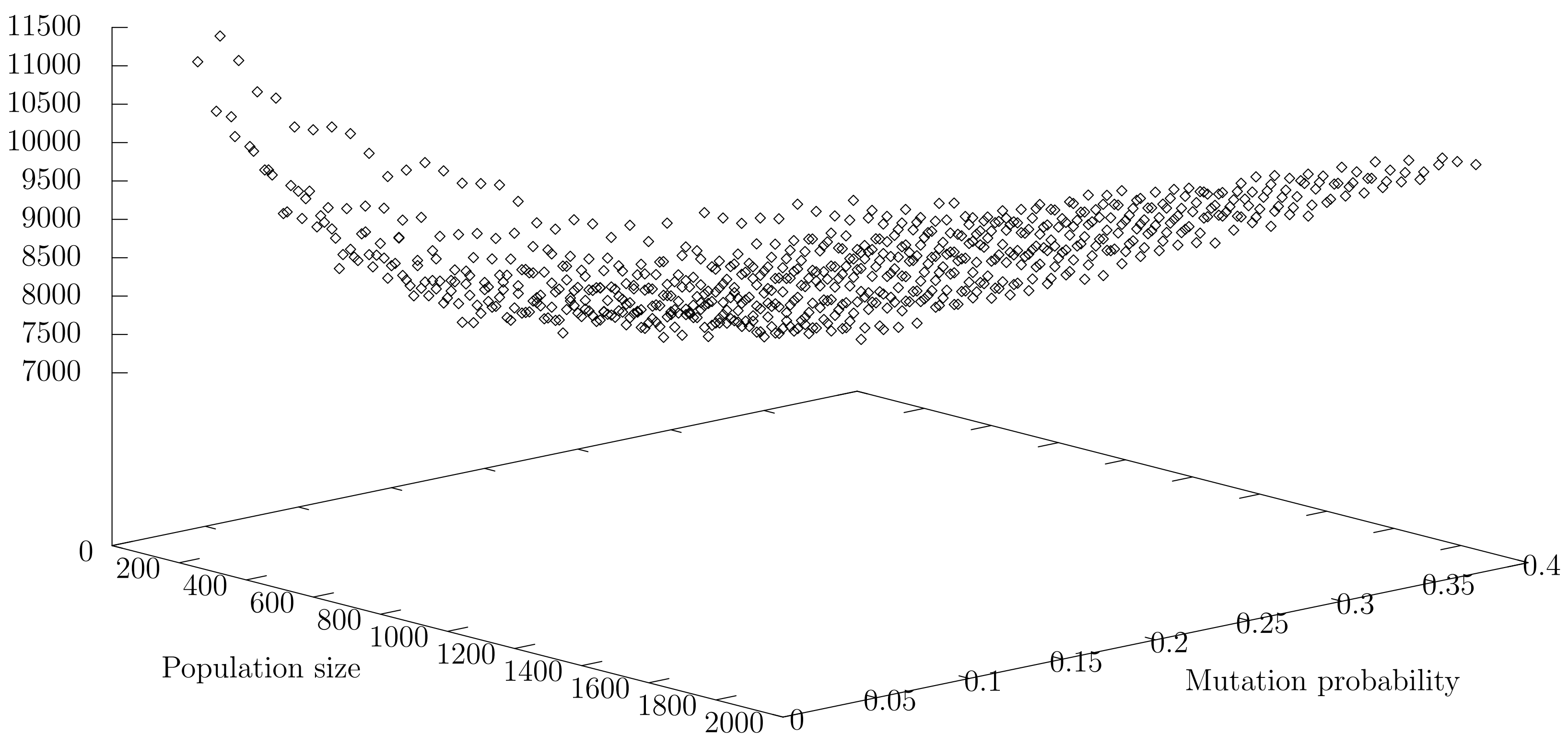


Figure 17: Population and mutation probability

2. Of two optimal solutions the one that was found with less reproductions is better.

A problem of this measure is that non-optimal solutions are always treated equally, no matter how good they are or how fast they have been found. Fixing this is quite difficult, and not that important since we can solve most of the problems in our set with an optimal solution quality.

For the parametrization as well as for the final results we use a set of 478 problems known as J30 in the literature and first discussed in the work of Kolisch et al. [17]. Since this set of problems is used in many other works, we can compare our results directly to those of others to get an idea of the performance of our algorithms.

We use a reasonable value for the population size of 600, since this looks good from figure 17 and is also the value used by other works for the same set (see [12]). With this value we tried out a range of mutation probabilities for the schedule chromosome as seen in figure 18. In order to create this figure the 478 problems in our test set were tried for mutation probabilities from 1% to 60& in 1% increments 12 times each.

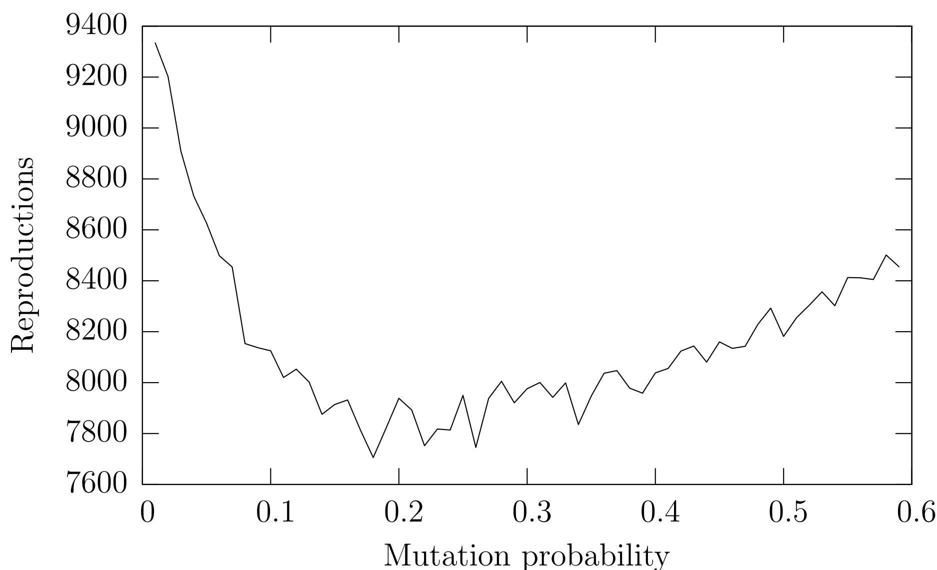


Figure 18: Mutation parameter (Population size = 600)

It is important to understand that the mutation rate does not state how often a mutation happens, but how often the mutation operator is tried. If you look at the definition of the mutation operator at the end of section 1.3.3

you will notice that it does not modify anything if the precedence relations do not allow this. It is hard to tell how often that happens in reality, but assuming random sequences 28% of all attempted mutations should fail for our problem set. Even taking this into account the optimal value of 0.2 is much higher than the one used by other implementations (0.05 in the case of [12]). This is probably due to the implementation of the mutation operator, which is not described in great detail in these works.

Using the value of 0.2 for the mutation probability, we try to optimize the population size as seen in figure 19 and end with the original value of 600. Since these values are similar to the ones used in the literature we can use them with reasonable confidence.

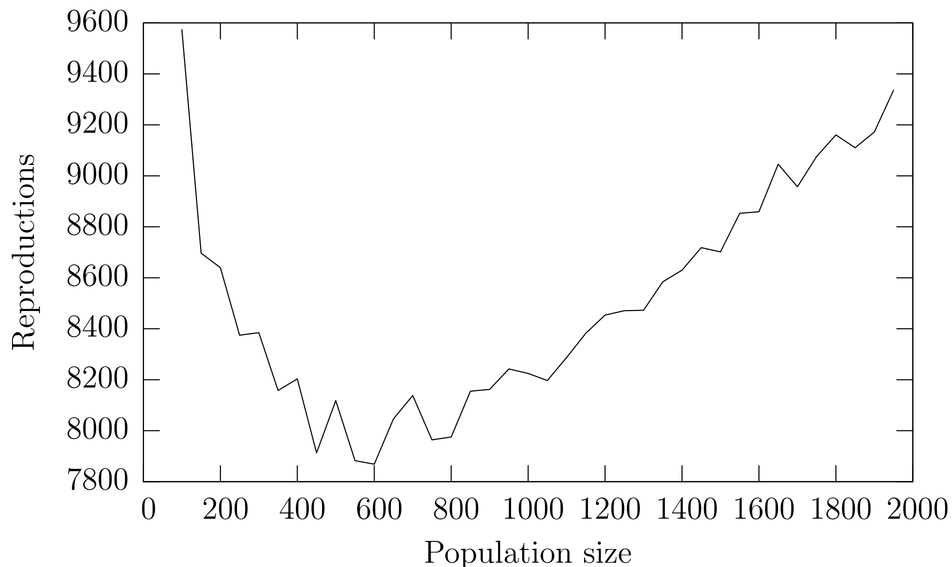


Figure 19: Population size (Mutation rate = 0.2)

When doing parametrization like this it is also very interesting how the optimal parameter setting looks for different instances in the problem set. If they are very different, one might want to look for some way to differentiate the problems and derive better parametrization values. For example the number of precedence relations in the problem could be used to modify the mutation parameter. If they are very similar a static parametrization looks reasonable. While this is very difficult we want to look at it anyway. In order to do so we first need to classify the problems according to their relevance. This is due to the fact that some problems are so trivial that they can be solved quickly even with sub-optimal parameter settings, whereas other problems are so difficult that we cannot solve them optimally no matter what param-

eter settings we use. Both of these types are not very interesting for us. We therefore take the data from the parametrization above and calculate a standard deviation for each set over the parameter used. If this value is high the parameter has a high impact on the performance of the algorithm for this problem and the problem is therefore significant. If the value is low the parameter has little impact on the algorithm performance and the optimal value for this problem is not very significant. For each problem we then calculate the optimal parameter value and use these two values to generate a diagram as in figure 20. Each point represents one problem, with the ones at the bottom being insignificant and the ones at the top being interesting. It's obvious that the problems at the bottom cluster around a mutation rate of 0.3 (the average of the whole parameter range) and spread out when you go up in the diagram. In the interesting part of the diagram above a significance value of 10000 most of the problems have an optimal value for a mutation probability in the range of 0.1 to 0.3. While this is still a wide range it basically looks pretty good for our parametrization considering that the performance beyond the used value of 0.2 declines only slowly. In any case it would be a very interesting research topic to look for problem metrics that correlate with these optimal values better. For the population size this method does not lead to reasonable results, which is due to the fact that a fixed part of the number of reproductions taken (the initial population) is created randomly and therefore contributes in a different way to the total performance. This leads to the fact that for easy problems the number of reproduction cycles that are done in the genetic algorithm phase differs very much compared to the number of reproduction cycles spent in the initial population generation phase. This leads to skewed results in the lower part of the diagram, as can be seen in figure 21. This is not that important as this is the insignificant part of the diagram anyway, in the significant part we can clearly see a concentration in the range of 400 to 1200. So in both cases the optimal parameter values differs between different problems, but only about a factor of 2 which we consider reasonable.

Since we know the optimal make-span of the problems used for the parametrization, we can also directly compare our results with it to get a measure on how good our algorithm performs. Using a maximum of 50000 reproductions, we get an average deviation from the optimal solution of 0.48 % with 80.2 % of all problems in the set being scheduled in an optimal manner. If we compare this with the results of other works for the same problem set (called J30, see 3.1.1) we see that we produce as good results as the best published algorithms, but need a lot more reproductions. This is due to the fact that we do not use many of the sophisticated techniques used by the

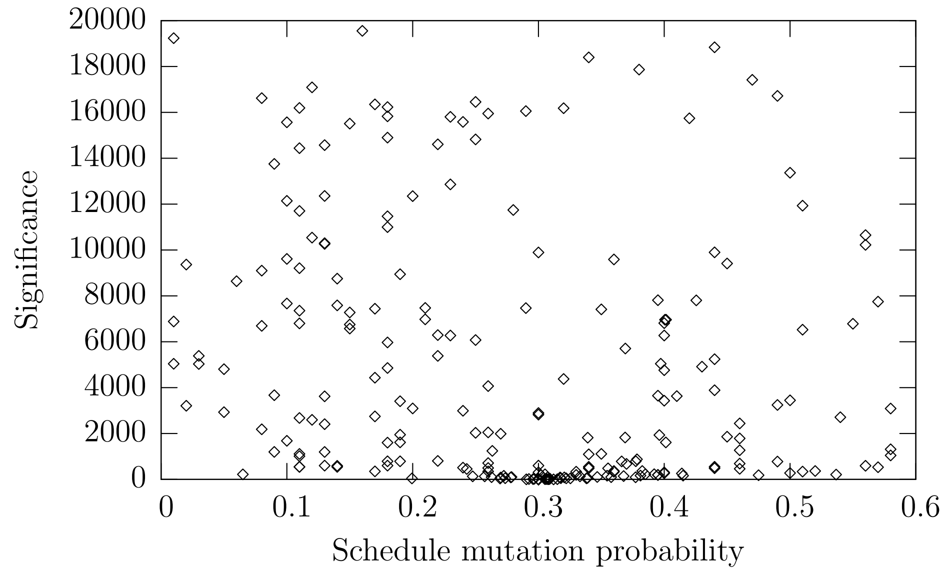


Figure 20: Distribution of optimal values for the mutation probability

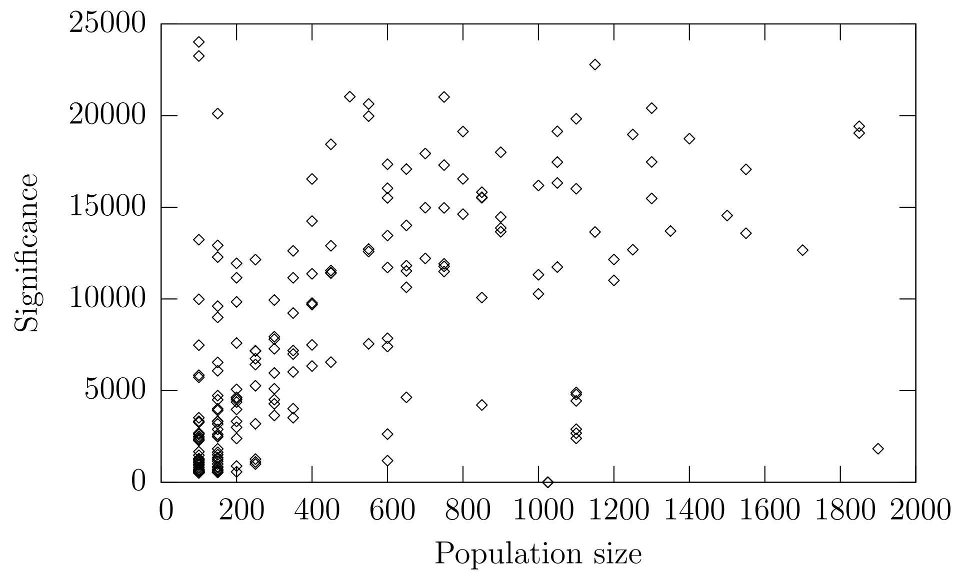


Figure 21: Distribution of optimal values for the population size

other algorithms, like an improved initial population through heuristics or adaptive genetic algorithms, yet. Anyway, the good results show that our parametrization is fine and we can use the algorithm to base our work on it. For reference S. Hartmann [12] reached 0.54 % average deviation with 81.5 % optimal schedules in 1000 reproductions, V. Valls et al. [24] managed an average deviation of 0.37 % in 100000 reproductions.

Applying the same parametrization scheme to the modes chromosome leads to surprises: we used a different set of problems that also employs multiple modes and tried many different mutation rates, with the results shown in figure 22. It is easy to see that this result is completely bogus and doesn't help us at all. Why this is the case could not be determined completely, but we assume that the low number of modes in the test cases is responsible for this. Since the mode genes are completely independent from each other, and the number of modes for a single job is much smaller than the population size, we will most certainly have all possible alleles for a single modes gene in the initial population. In this case it is possible to create every possible genome only using the crossover operator, so the mutation is not needed. This was not researched any further as we don't need optimal parametrization for this chromosome in this work anyway, as will be shown in section 3.2.

Because there exist no test cases with resource alternatives yet, we also cannot determine a good mutation rate for the corresponding chromosome. This is not a problem, as we will see that this does not have any impact on the outcome of this work since we can achieve good results even with sub-optimal parametrization.

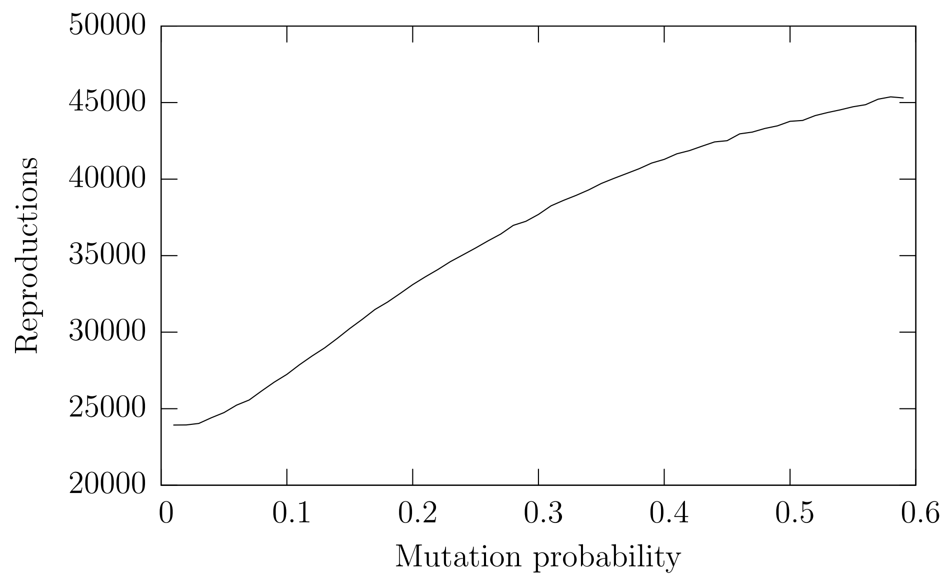


Figure 22: Mutation rate for the modes chromosome

3 Results

As we have discussed before, our proposed extension of the project representation allows us to model our project in a more natural and expressive way. But there is no problem that we can solve with our extension that could not be modeled before. So where is the gain? While we can model everything without our extension through an extensive usage of modes, we will get problems when that number of modes gets too large, because of time and memory constraints. And as we have seen before, the number of modes needed to model specific problems can be quite dramatic. So what we want to achieve is to be able to model and solve all the problems that could be solved without our extension and in addition be able to solve problems that would otherwise be difficult to solve because of the immense amount of modes needed.

In order to do so we first need to create test cases for these problems, and then try to solve them with and without our extension. Since we currently only look at the number of reproductions needed to find a solution of a given quality, we are in danger to trade a benefit there for a drawback in actual computation time. We will therefore also show that our extension adds no significant computational overhead to the algorithm.

3.1 Testing Method

3.1.1 Test Cases

As stated before we use a set of problems called J30 as the basis of our tests. These are widely used in the literature and allow us to not only compare our extension with our code without the extension, but also with the results of other people.

3.1.2 Transformations

All problems in the set are single-mode problems that we will use to create the actual test problems from. We will use transformations to first create problems using resource alternatives from them and then to create a corresponding problem that uses only modes from it. After applying these we will have three different problems: a single-mode problem, a problem using resource alternatives, and a problem with multiple modes. It is important to note that all three of these actually describe the same project and therefore

have the same optimal make-span. They do however encode the project in different ways with different complexities, it is therefore unlikely that we will see as good a performance on the latter two problems as on the first one.

Transformation one will take each resource k_i with a capacity of r_i and create r_i additional resources with a capacity of 1 each that represent the individual elements of this resource. So if we would have a resource "Worker" with a capacity of 4, our transformation would turn this into 5 resources: the original "Worker" resource with a capacity of 4 and "Worker1" through "Worker4" with a capacity of 1 each. This way we now can identify each individual resource element. In a second step we will modify the resource requests on the jobs: we choose a number N that denotes how many individual resources we want to use on each job and then add up to N or the amount requested resource requests to each resource request already present. In these additional resource request we use r_i resource alternatives (with a capacity of 1) on the new resources that correspond to the resource elements of the resource request already there. In the example above we could have a job that requires 2 workers. For $N = 3$ we would have to create 2 additional resource requests which would each have the same 4 resource alternatives: "Worker1" through "Worker4" with an request amount of 1 each. Please note that if N is less that the original request amount, not all of the resource elements are selected individually. This does not change the basic project however, because we still have the request on the original resource that now functions as a "resource group". Since this transformation is fairly complex, figure 23 gives another example for a project before and after the transformation.

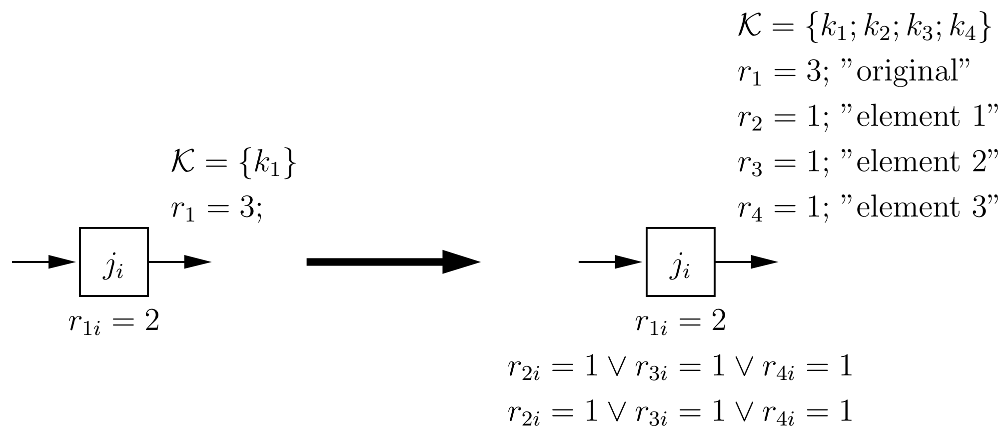


Figure 23: Example for transformation 1

The second transformation will modify a project that uses resource alternatives into one that is identical in outcome but uses only modes to represent

the different possibilities. This is carried out by taking each mode (in our case always one) of the input project and create M modes for it, where $M = \prod_R A_R$. This means we take every possible combination of resource alternatives for that mode and turn it into an own mode with only one resource alternative per resource request. To avoid modes that are impossible to schedule, and that were possible in the previous representation due to the repair operator, we have to remove all modes where the number of resource requests for a single resource exceeds its capacity. In the example with the four workers above we would turn the job with the two resource requests that have the four workers as alternatives into one job with 16 initial modes (all combinations of the first resource request and the second resource request in the previous representation) of which we can remove 4 because they exceed the resource availability (every time the first and the second resource request use the same worker).

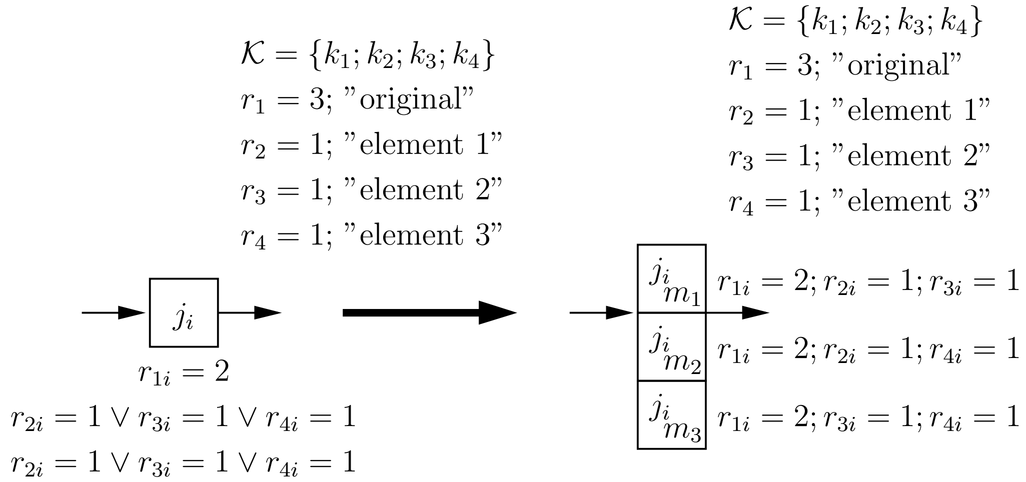


Figure 24: Example for transformation 2

While these transformations are fairly complex and a bit difficult to understand, the important properties are quite simple:

- We use existing problems as a basis, which is good because we can then use the trusted J30 problem set
- We can create derived problems that sport a tunable amount of resource alternatives
- And we can create representations of the derived problems in a form that does not use resource alternatives but only modes

It is therefore quite easy to compare the performance of our extended algorithm with the existing solutions.

3.1.3 Test Setup

Since these tests are quite elaborate, a robust test setup was needed in order to create reproducible and, more importantly, traceable results. In our case we always use an SQL database that stores which test case was run with which parameters and what result along with additional information like the time of the test run and the binary check-sum. This database is filled by perl scripts that run the program binary on the test cases. The transformed test cases were created by special functions in the binary and saved on disc under special directories. Of course each test was run multiple times because of the probabilistic nature of the algorithms.

The same setup was used for the parametrization and proved to be very scalable and flexible. It is quite easy to extract the information and even possible to use multiple machines for the test runs.

3.2 Empirical Results

When running the tests, we first chose a very conservative value for N of 1, which means that for every resource request only one resource element is being identified individually. During the first transformation (traditional model to resource alternatives) we could easily transform all 478 problems. The total size on disc rose from 6 megabytes to 54 megabytes, due to the extended problem space. Please keep in mind that the size of a problem in an XML file is much bigger than it is once it is in the internal representation in memory, we just give these figures to get a picture of the dimensions we are dealing with. The second transformation proved to be much more difficult. Because of the combinatorial explosion of the number of modes, not all problems could be transformed. With a limit of 256 megabytes on the amount of memory given to each process we could transform only 262 of the 478 problems which resulted in a total of 14 gigabytes of problem descriptions. The corresponding problems 262 problems after the first transformation take up about 16 megabytes. So a first conclusion, even before running any test, is that even with very low N settings only a limited subset of our test cases can be expressed with multiple modes without getting excessively large.

When running our program on these 262 problems in all three variants, we see that these are easy to solve in the original form, and still fairly easy after

the first transformation. Some of them could not be solved due to memory constraints, but most perform fairly good. Of the 262 problems 235 (89%) could be solved resulting in an avergage deviation from the optimal solutions of less than 0.1%. In the full set of the 478 problems, 376 (79%) could be solved, also with an average deviation of less than 0.1%. The problems that could not be solved here failed due to extreme memory requirements of more than 1 gigabyte. It is also interesting to note that we did not do a proper configuration of the mutation rate of the resource alternatives chromosome due to the problems outlined in section 2.2.6, but just assumed a value of 0.05. So there might be room for improvements here. The fact that we cannot solve all problems after the first transformation is of no big concern, as this transformation actually adds complexity to the project itself. The problems that could not be solved are (as determined by some random samples) all problems with very large capacities on the resource, which in turn leads to a large number of alternatives. Also interesting is the low average deviation in the transformed sets. Since we removed the problems we could not solve at all, this means that the problems that did produce sub-optimal results in the original form are also the ones that could not be solved at all after the transformation. After the second transformation we only have the 262 problems left, and out of these 155 (59%) could be solved, showing an average deviation of 0.3%. The original problems could all be solved with an average deviation of 0.48%. If we only look at the 262 problems that were used after the second transformation, we get an average deviation that is almost 0%.

While using a value for N of 2, we can still apply the first transformation to all problems, resulting in 93 megabytes of problem descriptions. Only 120 of the problems could also be transformed to a mode-based representation due to the high memory requirements. These 120 problems take up 194 megabytes in the third form and only 6.9 megabytes in the second one (a clear indication that only the smallest problems could be transformed). These remaining problems can be solved with a average deviation from the optimal solution of almost 0% in all three forms. If we look at the problems in question we quickly find out why: these problems are very simple from a resource-centric point of view. This means that they contain very few resources with a low capacity and only very few resource requests per job. If you look at the definition of our transformations above, you will see that problems with these properties will not get more complicated with increasing values for N if N is larger than the maximum of the three properties given above.

Looking at the data above the set of problems can be divided into three groups: a small proportion of the problems cannot be transformed into difficult problems because of their structure and can therefore be solved easily

in any of the three forms. This is a problem of test setup and the transformations involved. Another small proportion of the problem set gets extremely complicated when transformed and therefore cannot be transformed and solved successfully. This is also a problem of our test setup. Most of the problems however can be transformed and solved when using our extension, but either are impossible or extremely hard to solve because of their increased size. In no case did our proposed extension perform worse than the traditional approach.

3.3 Time and Space Complexity

The performance metrics we gathered above deal with the number of reproduction cycles needed to reach a solution of a given quality, but it is also important to check the complexity of a single reproduction cycle. In order to do so it is very useful to separate the program into different parts.

First of all there are the problem-independent parts of the program, like the population management. These are by nature not affected by our changes but we tried to make them as efficient as possible anyway. For example a specialized AVL tree, that has $\mathcal{O}(1)$ access and deletion characteristics for the first and the last element on top of the $\mathcal{O}(log(n))$ access complexity for insertion, was used to implement the population container.

Then there are problem-specific parts that were not changed, like the crossover and mutation operators for the schedule and modes chromosome. We tried to get good performance here too (for example through caching precedence relations that are needed for the schedule mutation operator), but this does not change the basic fact that these operations do not differ from the traditional approach.

Some of the parts that are different between our solution and the traditional one are of little concern because they are only called a limited number of times, like during initial population generation. While this theoretically might still be a problem, it is not in our case. Since the only difference is that we have to fill an additional chromosome with random data, our time complexity is still $\mathcal{O}(n)$ where n is the number of modes plus the number of resource alternatives. The initial generation of the schedule chromosome is the same as before. This is an interesting point however, as this might change drastically if we move to an improved generation of the initial population that uses heuristics, which has been done with quite some success in the literature.

Likewise the crossover and mutation operators for the resource alternatives chromosome are the same as those for the modes chromosome and have the same complexity. Since our extension is intended to reduce a massive amount of modes to a reasonable amount of resource alternatives, we should actually benefit from our changes in these areas.

Only two parts are of bigger interest: the repair operator and the decoding function. Since we use a very simple repair operator, we can answer this quite easily: our repair operator has a (worst case) time complexity of $\mathcal{O}(n)$ where n is the total number of resource alternatives for the project. If we start using more powerful repair operators this will become increasingly interesting. Please note that the repair operator is only called in very few cases (3% in our tests).

The decoding function is the most interesting part here, since our application spends the most time (65% in example cases) there. The time complexity of the basic SGS is known to be $\mathcal{O}(J^2 \cdot K)$ where J is the number of jobs and K the number of resources (see Pinson et al. [19]). This does not change by adding multiple modes or resource alternatives as these are not relevant to the SGS and just show up as simple redirections with a $\mathcal{O}(1)$ time complexity.

Our space complexity before was a linear combination of the number of jobs, modes and resources used. In our new representation we have to add the alternatives as well, but we are not adding any more complex data structures. Basically space is not an issue as long as the number of entities in our model stays in a reasonable range, which is what we are trying to ensure with our extension.

4 Discussion

4.1 Achieved Goals

Originally, the intention when starting this work was to create a way to handle a specific problem with the traditional approaches to the resource constrained project scheduling problem: we cannot model "or" relations on the resource request level. While this sounds like a minor problem, it should become obvious when reading section 1.4.5 that this is an important feature for many projects. Traditionally this would have been solved with the multiple modes extension.

In this work we managed to explain and show that modeling these cases with multiple modes is, while theoretically possible, unfeasible in practice due to a possible combinatorical explosion in the number of modes needed.

An additional extension was proposed, the resource alternative. This extension adds another degree of freedom, much like the multiple modes, on the resource request level. This extension was designed, implemented and parametrization values determined as far as possible and necessary. The algorithm turned out to perform well, despite the fact that many improvements like adaptive algorithms with multiple decoding functions (see [11]), complex operators (see [24]), improved initial populations (see [12]) or a local search phase (see [12]) were not used. It is very likely that our algorithm will perform as well as any of the currently known ones if these improvements were added, but even without them we can schedule standard projects very well.

We have shown that many problems that can easily be solved with our extension are impossible to solve with the traditional approach when the computing resources are limited. The corresponding problems modeled with mutiple modes also proved to be very cumbersome to handle due to their extreme size (see section 3.2), another benefit of our extension.

Since our representation is a super-set of the resource constrained project problem with multiple modes, which in turn is a super-set of the original resource constrained project scheduling problem, we can handle these simpler cases with our algorithm without modification. The time complexity of our algorithm in comparison to the original one was analyzed to show that we don't change the run-time behavior of the algorithm for a single reproduction cycle (see section 3.3).

4.2 Conclusion and Future Work

Overall we are confident that the proposed extension of the problem representation allows for a greater flexibility and expressiveness when modeling projects. More importantly, our extension avoids the combinatorical explosion of the number of modes needed that could arise in some projects and makes them hard or impossible to solve. All problems that can be solved with the traditional representation can also be solved with our extension with comparable performance. We therefore think the proposed extension can be recommended to any application dealing with non-trivial project scheduling problems. However, A couple of interesting topics have surfaced during this work that should be researched in greater depth:

Due to the way the testing procedure was set up, we could not determine proper configuration parameters for the mutation rate of the modes and the resource alternatives chromosome. While we could easily produce the intended results with guessed and therefore probably sub-optimal parameters, it would still be very interesting to determine optimal values for these parameters.

Another problem is that we created our test cases through transformations from original test cases, and are therefore not able to create arbitrary problems. A better set of test problems would be very desirable, and would also allow us to derive parametrization values for the resource alternatives chromosome.

The repair operator used in this work is pretty simple in it's current form, different possibilities to implement it should be tried and compared. It would also be interesting to find out how complex such an operator may be to achieve optimal performance.

On our resource alternatives chromosome we encounter genes that are not evaluated in the decoding function because the belong to a mode that is currently inactive. It would be interesting to find out whether masking out these genes from the mutation operator improves performance or not.

Many of the improvements made in the literature can be applied unmodified to our algorithm as well.This should be done to improve the overall performance of the algorithm. This includes improved operators (see [24]) and adaptive algorithms with different decoding functions (see [12]).

Improved initial population generation through heuristics also falls into this area, but is a bit special since it cannot be applied unmodified. Instead heuristics for the resource alternatives have to be developed, which might

prove difficult. It should however be tried whether only applying the heuristics to some chromosomes is already beneficial.

Also in this area is the usage of a local search phase. While this has been shown to improve performance in the literature, we very much doubt that this is true if you measure performance not in terms of reproduction cycles, but in terms of real-life computation time.

Finally, our extension only goes a first step in the direction of making the problem representation more flexible. Any research in that direction should be interesting.

List of Figures

1 Example project 8
2 Example project schedule 8
3 Example problem function 13
4 Example crossover operator 14
5 Example chromosomes 17
6 One-point crossover for permutation-based genotypes 18
7 Two-point crossover for permutation-based genotypes 18
8 Example multi-mode project 21
9 Example multi-mode project schedule 21
10 Multi-Mode genome 23
11 One-point crossover for the modes chromosome 23
12 Two-point crossover for the modes chromosome 23
13 Example project with resource alternatives 27
14 Example project schedule with resource alternatives 27
15 Resource alternatives genome 29
16 Example chromosome 30
17 Population and mutation probability 35
18 Mutation parameter (Population size = 600) 36
19 Population size (Mutation rate = 0.2) 37
20 Distribution of optimal values for the mutation probability . . 39
21 Distribution of optimal values for the population size 39
22 Mutation rate for the modes chromosome 41
23 Example for transformation 1 43
24 Example for transformation 2 44

References

[1] Alvarez-Valdés and J. Tamarit. Heuristic algorithms for resource-consrained project scheduling: A review and an empirical analysis. *Advances in project scheduling*, 1989.

[2] F. F. Boctor. Some efficient multi-heuristic procedures for resource-constrained project scheduling. *Europeal Journal of Operations Research*, 1990.

[3] D. Cooper. Heuristics for scheduling resource-constrained projects: An experimental investigation. *Management Science*, 1976.

[4] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics*, 1996.

[5] S. E. Elmaghraby. *Activity networks: Project planning and control by network models.* Wiley, 1977.

[6] M. Garey, R. Graham, D. Johnson, and A.-C. Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory*, 1976.

[7] M. Garey and D. Johnson. Computers and intractability: A guide to the theory of np-completeness. *Freeman*, 1979.

[8] F. Glover. Tabu search – part i. *ORSA Journal on Computing*, 1989.

[9] F. Glover. Tabu search – part ii. *ORSA Journal on Computing*, 1989.

[10] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, 1989.

[11] S. Hartmann. A competitive genetic algorithm for resource-constrained project scheduling. *Naval Research Logistics*, 1998.

[12] S. Hartmann. *Project scheduling under limited resources: models, methods, and applications.* Springer-Verlag, 1999.

[13] H. J. Holland. *Adaption in natural and artifical systems.* University of Michigan Press, 1975.

[14] J. E. Kelley. Critical-path planning and scheduling: Mathematical basis. *Operations Research*, 1961.

[15] S. Kirkpatrick, J. C.D. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 1983.

[16] U. Kohlmorgen, H. Schmeck, and K. Haase. Experiences with fine-grained parallel genetic algorithms. *Annals of Operations Research, Volume 90*, 1999.

[17] R. Kolisch, A. Sprecher, and A. Drexl. Characterization and generation of a general class of resource-constrained project scheduling problems. *Management Science*, 1995.

[18] J. Patterson. A comparison of exact approaches for solving the multiple constrained resource, project scheduling problem. *Management Science*, 1984.

[19] E. Pinson and C. P. adn F. Rullier. Using tabu search for solving the resource-constrained project scheduling problem. *Proceedings of the fourth international workshop on project management and scheduling*, 1994.

[20] C. Reeves. Genetic algorithms and combinatorial optimization. *Applications of modern heuristic methods*, 1995.

[21] L. Schrage. Solving resource-constrained network problems by implicit enumeration - nonpreemptive case. *Operations Research*, 1970.

[22] A. Sprecher. Resource-constrained project scheduling: Exact methods for the multi-mode case. *Lecture Notes in Economics and Mathematical Sytstems Number 409*, 1994.

[23] A. Sprecher and A. Drexl. Multi-mode resource-constrained project scheduling by a simple, general and powerful sequencing algorithm. *OR Spektrum*, 1984.

[24] V.Valls, F. Ballestin, and S. Quintilla. A new crossover operator for the resource constrained project scheduling problem. *MIC2003: The Fifth Metaheuristics International Conference*, 2003.