Analysis and Improvement of Constraint Handling in Ant Colony Algorithms

by

Martin Held, B.Sc.

Thesis
Submitted by Martin Held
in partial fulfillment of the Requirements for the Degree of Bachelor of Computer Science with Honours (1608)

Supervisors: Dr. Bernd Meyer and Dr. Andreas Ernst

Clayton School of Information Technology
Monash University
November, 2005
© Copyright
by
Martin Held
2005
Contents

List of Tables ................................................................. v

List of Figures ................................................................. vi

Abstract ................................................................. viii

Acknowledgments .......................................................... x

1 Introduction ................................................................. 1
  1.1 Problem Domain ......................................................... 2

2 Background ................................................................. 3
  2.1 Combinatorial Optimization Problems ................................. 3
  2.2 Systematic Approaches .................................................. 4
    2.2.1 Introduction ......................................................... 4
    2.2.2 Constraint Programming .......................................... 4
  2.3 Incomplete Algorithms/Stochastic Search .............................. 6
    2.3.1 Introduction ......................................................... 6
    2.3.2 Metaheuristics ...................................................... 7
    2.3.3 Constraint Handling ............................................. 10
  2.4 Constraint Programming and Metaheuristic Hybrids .................. 12
    2.4.1 CPACS ............................................................. 13
    2.4.2 Local Search and Constraint Programming ......................... 13

3 Stochastic Ranking and ACS ............................................ 15
  3.1 Introduction .......................................................... 15
  3.2 Methods ............................................................... 16
    3.2.1 Testing Procedure ............................................... 16
  3.3 Results ............................................................... 16

4 Constraint Programming and ACS ...................................... 19
  4.1 Introduction .......................................................... 19
  4.2 Methods ............................................................... 23
    4.2.1 Hybridization of CP and ACS .................................... 23
    4.2.2 Domain Filtering ............................................... 25
    4.2.3 Dynamic Heuristic Values ..................................... 29
    4.2.4 Coupling CPACS and ACSsr ..................................... 30
  4.3 Results ............................................................... 31
    4.3.1 CPACSlp ........................................................ 32
    4.3.2 CPACSlp without learning component ............................ 34
    4.3.3 Effect of Dynamic Guidance .................................... 35
    4.3.4 Coupling CPACS and ACSsr ................................... 37
5 Conclusions

5.1 Future Research

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>Labeling Steps – Computation Time Tradeoff</td>
<td>40</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Tightness of Bounds</td>
<td>40</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Efficient Relaxation Solving</td>
<td>40</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Insertion of Exchanged Solutions</td>
<td>40</td>
</tr>
<tr>
<td>5.1.5</td>
<td>More Learning</td>
<td>41</td>
</tr>
</tbody>
</table>

Appendix A Test data

Appendix B Obstacles

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>Problem with all different Constraint / Weak Solver</td>
<td>45</td>
</tr>
<tr>
<td>B.2</td>
<td>Severe Cluster Instability</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Performance Results for ACScion and ACSsr ........................................ 18
4.1 Solution values until labeling step were counted ................................. 32
4.2 Results of CPACS – learning, no bounding and CPACS – learning, bounding
(50 independent runs) ................................................................. 33
4.3 t-Test results (p-Values) .............................................................. 34
4.4 Results of CPACS – no learning, no bounding and CPACSlp – no learning,
bounding (50 independent runs) .................................................. 34
4.5 t-Test results (p-Values) .............................................................. 35
4.6 CPACSlp – Dynamic Heuristics vs. Static Heuristics ............................ 36
4.7 t-Test results (p-Values) .............................................................. 36
4.8 CPACS + ACSsr – Makespan results ($pf = 0.7$, 20 independent runs for
CP(ACS)$^2$sr), Results for CPACS and ACSsr taken from above, 50000 tour
constructions in ACSsr, 10 independent runs for ACSsr .................. 37
# List of Figures

1.1 Single Machine Job Scheduling .................................................. 2

2.1 Shortest path finding behaviour ................................................. 8

2.2 Stochastic Ranking procedure, $I_j$ represent a solutions/individual with index $j$ .................................................. 12

4.1 Search Tree of an CP Assisted Ant - Numbers at edges represent traveling time between cities, pair of numbers at nodes represent time windows in which the respective cities have to be visited (#release, #deadline), dashed lines mark possible paths ruled out by constraint propagation ........................................... 20

4.2 TSP Relaxation – The graph on the left hand represents the assignment problem. Each city $i$ must be assigned a city $j$ which represents its successor. The graph on the right hand side is a more convenient representation of the same information. The edge costs are not annotated in the AP graph but are the same as in the graph on the right. Bold edges represent realized edges. Figure (a) represents the optimal solution if the problem would be considered as a real TSP, whereas Figure (b) displays the optimal solution of the AP (sub-tours are allowed) ........................................... 27

4.3 Coupling of CPACSlp + ACSsr – interleaving scheme .................. 31
List of Algorithms

1. Basic ACO algorithm .................................................. 9
2. ACSsr: ACS in Combination with Stochastic Ranking .............. 17
3. ACScoin: Replacement for ACSsr lines 35-37 ....................... 18
4. Basic CPACS – combination of CP and ACS ....................... 25
Analysis and Improvement of Constraint Handling in Ant Colony Algorithms

Martin Held, B.Sc.
Monash University, 2005

Supervisor: Dr. Bernd Meyer and Dr. Andreas Ernst

Abstract

Ant Colony Optimization (ACO) is a metaheuristic which has been successfully applied to tackle various combinatorial optimization problems (COPs), but its ability to cope with hard constrained COPs is yet to be explored widely. Since most real world optimization problems, e.g., job scheduling, rostering or timetabling are subject to hard constraints, the investigation of different constraint handling methods within ACO becomes an important area of study. We explore the applicability of Stochastic Ranking as a soft handling technique and propose a bounding scheme as an extension to an existing ACO implementation, that handles constraints in a hard fashion, by utilizing Constraint Programming. Based on this extension we suggest a new search guiding approach that replaces static heuristic values by dynamically calculated values. Moreover, we attempt a loose coupling of soft and hard constraint handling techniques in one algorithm. Conducted empirical studies showed that the calculated upper bound on the solution quality is not tight enough to significantly improve the performance of the existing algorithm. In consequence, we also did not observe a significant performance enhancement for the other attempted extensions.
Analysis and Improvement of Constraint Handling in Ant Colony Algorithms

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Martin Held
November 11, 2005
Acknowledgments

First of all, I am indebted to Dr. Bernd Meyer and Dr. Andreas Ernst for giving me the opportunity to work in such an interesting research area. They patiently answered all of my questions and kept motivating me throughout the project.

Additional thanks go to all people who supported my during this interesting journey, especially to Mira and my parents.

Martin Held

Monash University
November 2005
Chapter 1

Introduction

Combinatorial optimization problems (COPs) occur in many industrial and scientific areas. Solving them is often difficult as most of them are NP-complete. Different approaches have been developed to tackle them. They can be divided into complete and incomplete approaches. Complete approaches search the solution space exhaustively and therefore guarantee to find the optimal solution. On the other hand, incomplete methods perform a non-exhaustive search by cleverly sampling parts of the solution space. Due to this incompleteness they cannot guarantee to find the optimal solution.

The exhaustive search property of complete methods leads in the worst case to an exponential time complexity, whereas incomplete methods are likely to find good quality solutions in a reasonable amount of time.

A representative for an incomplete method is Ant Colony Optimization (ACO). It is a nature inspired metaheuristic which is based on shortest path finding behaviour of ant colonies. ACO has been successfully applied to tackle combinatorial optimization problems of various kinds [Dorigo et al., 1999, Cordon et al., 2002].

However, most real world COPs, e.g., job-scheduling, timetabling or rostering are subject to hard constraints. These constraints pose difficulties to the utilized search algorithms. The solution space becomes fragmented and the question of handling infeasible solutions arises. How should solutions that violate a constraint be used during the search process to eventually guide to better feasible solutions or should they better avoided completely? Constraint handling in metaheuristics, and in ACO in particular, is an open area of research.

In this thesis we investigate different ways of constraint handling within an Ant Colony Optimization. The first part is concerned with the analysis of the applicability of Stochastic Ranking (SR) in ACO algorithms. SR is a soft constraint handling technique that has been developed in the context of evolutionary optimization and successfully applied to ACO. We investigate how its mechanisms works in the ACO context and examine if it might be replaced by a simpler procedure.

In the second part we suggest a number of extensions to an ACO algorithm that uses constraint programming (CP) techniques to handle constraint violations in a hard way. We attempt to increase the lookahead that is provided by the CP system by incorporating extra information gained from solving a relaxation of the problem. Furthermore, we utilize this information to develop the principle of dynamic guidance. An attempt to replace static problem depended heuristic values with values that change with the structure of the surrounding search space. Finally, we investigate if a loose coupling of soft and hard constraint handling techniques is worthwhile.

To asses our developed approaches we apply them to a set of single machine job scheduling problems. The comparison is based on the achieved solution quality and the amount of search that is carried out.
1.1 Problem Domain

A single machine job scheduling problem consists of finding an ordering of a number of jobs such that the time needed to process all these jobs (makespan) on a single machine is minimal. Each job has an assigned duration, a release time and a due date, where duration is the time the job needs to be processed by the machine and the release and due times specify a time window in which the job can be started and must be completed. Furthermore, there is a setup time assigned with each pair of jobs that models the time it takes to setup the machine to change from processing one job to another.

Single machine job scheduling problems can also be regarded as asymmetric traveling salesman problems with time window constraints (ATSP-TW). Our test cases are described in more detail in Appendix A. Figure 1.1 shows a possible schedule of three different jobs. The brackets indicate the possible time the respective job can be scheduled. The time windows of job A and B for example would allow to schedule A after B, assuming the setup time from B to A would together with the duration of A not violate A’s due date constraint. The figure also gives an example for possible introduced waiting time for job C. The makespan for the displayed assignment would be calculated by $\text{makespan} = \text{duration}_a + \text{setup time}_{a,b} + \text{waiting}_{a,b} + \text{duration}_b + \text{setup time}_{b,c} + \text{waiting}_{b,c} + \text{duration}_c$, where $\text{waiting}_{a,b}$ is 0 in this particular example.

![Figure 1.1: Single Machine Job Scheduling](image-url)
Chapter 2

Background

2.1 Combinatorial Optimization Problems

Solving optimization problems consists of finding an optimal assignment of variables such that a certain goal is achieved. Optimization problems can be divided into two categories: problems dealing with continuous variables and problems concerned with discrete variables. Optimization problems involving discrete variables are also called Combinatorial Optimization Problems (COPs) [Papadimitriou and Steiglitz, 1982]. Solving continuous problems means to search for a set of real numbers or a function, whereas in combinatorial problems the search is concerned with finding the best solution out of a countable set, for example a set of integers.

Combinatorial Optimization Problems are very important as they have many industrial applications, for example scheduling, timetabling, rostering or vehicle routing. Probably the most famous instance of a COP is the Traveling Salesman Problem (TSP) [Lawler et al., 1985]. The problem consists of finding an optimal length tour through a number of cities, such that each city is visited exactly once.

Many algorithms have been proposed to tackle COPs. They can be divided into two different classes: complete and incomplete algorithms. Complete algorithms perform a complete search on the solution space and have the property that they guarantee to find the global optimal solution for a Combinatorial Optimization Problem of finite size in bounded time [Papadimitriou and Steiglitz, 1982]. As most CO problems are NP-hard [Garey and Johnson, 1979] this means, assuming $P \neq NP$, they are very hard to solve. Using a complete algorithm to solve a COP might therefore take exponential time, which is often inappropriate for real world problem instances. Incomplete techniques on the other hand perform only an incomplete search and cannot guarantee to find an optimal solution but are capable of finding near optimal solutions in a reasonable amount of time.

A special form of combinatorial optimization problems are constraint combinatorial optimization problems. In these problems the plain COP is additionally subject to side constraints. An example is the TSP with time windows. Here the objective is not only to find the shortest tour, but also to meet the time window constraints associated with each city. It is important to note that almost all real world optimization problems are subject to constraints. Optimization algorithms must therefore be capable of handling them. Constraints on the problem fragment the solution space and therefore impose difficulties for the search algorithms. It becomes difficult to generate feasible solutions and to move between different feasible areas in the solution space.

The following text is organized as follows: The first section discusses systematic-complete approaches to solve COPs, particularly Constraint Programming. Next, stochastic search techniques, in particular metaheuristics, are introduced. Within that context different methods of handling constraints within metaheuristic frameworks are explained. Finally,
the last section deals with various possibilities of extending meta heuristics with constraint programming as constraint handling technique.

2.2 Systematic Approaches

2.2.1 Introduction

Systematic or complete approaches have the property that they exhaustively explore the complete search space of a COP. This theoretically guarantees to find the optimal solution as all possible solutions are considered. Unfortunately, finding the optimal solution might take inappropriately long because almost all interesting COPs are usually NP-hard. However, using clever implementations this runtime can be reduced to be polynomial in the average case for some problems. Which, after all, makes complete approaches applicable for certain problems. Commonly used systematic methods include Linear Programming [Danzig, 1963], Integer Programming [Garfinkel and Nemhauser, 1972] and Constraint Programming.

2.2.2 Constraint Programming

Constraint programming (CP) is declarative programming paradigm that allows a programmer to formulate a problem in terms of variables, domains of variables and constraints between variables. In contrast to usual programming languages, the programmer has only to provide the problem representation rather than to explicitly specify the steps of solving it. A comprehensive survey of CP is given in [Marriott and Stuckey, 1998].

The earliest CP system was Sketch Pad [Sutherland, 1964] an interactive drawing tool. Later on CP techniques made their way into Logic Programming languages. Logic programming is also a declarative language that allows us the formulation of problems in terms of logical rules. It uses simple resolution algorithms to resolve the logical statements of a problem. The extension with CP techniques adds more powerful algorithms to resolve this statements.

Early constraint logic programming languages were for example CHIP [Hentenryck, 1989] and Prolog III. However, more recently constraint programming features became also available for C++ (ILOG-Solver [Solver, 2003]) and Java (JSolver).

Solving a CP problem means to find an instantiation of all problem variables in a way that all given constraints are fulfilled. This task is performed behind the scenes by a constraint solver. To obtain a valid instantiation the solver performs a constraint propagation step and a search over all possible variable assignments.

In the propagation step all given constraints are evaluated and the domains of variables are reduced depending on possible constraint violations. To find a feasible instantiation of each variables the solver performs a search by labeling a variable by selecting a value from its domain. This instantiation may cause some constraints to become violated such that domain values in other variables become infeasible. The violated constraints would propagate and automatically remove these infeasible values from the respective domains.

If during the search a value assignment leads to an error, meaning a domain of a variable becomes empty, the solver has encountered a dead end. In this case this particular variable assignment is not a valid solution. The last variable assignment is reset and the next possible value from the domain of that variable is tried. The whole process is called backtracking and described in more detail in the next section.

To illustrate the concept of constraint propagation we present the following example:
Problem representation

\[ X \in \{5, 23, 28, 35\} \]
\[ Y \in \{10, 42\} \]
\[ Z \in \{42, 84\} \]
\[ X \leq Y \]
\[ Y \neq Z \]

Suppose the value 23 is randomly chosen for variable \( X \). As the domain of \( X \) is changed, constraints associated with \( X \) will be checked. The constraint propagation will reduce the domain of variable \( Y \) to \{42\} because of the constraint \( X \leq Y \). This will cause the domain of \( Z \) to be reduced to \{84\}, because of the constraint \( Y \neq Z \).

Constraint Programming can be very effective in solving strongly constrained problems, because constraint propagation allows to prune the search space effectively. However, if a problem is only weakly constrained almost a complete search will be performed which is inefficient. An example of how CP can be used to solve TSPs with time windows is given by Pesant et al. [1998].

In the next two subsections two algorithmic methods (Backtracking and Branch-and-Bound) are explained as they are commonly used in CP frameworks.

**Backtracking**

Backtracking is the standard search procedure used in conjunction with CP to tackle combinatorial optimization problems. It is a procedure that systematically enumerates all potential solutions of a problem without enumerating a solution more than ones. Early descriptions of backtracking can be found in Walker [1960] and Colomb and Baumert [1966]. To be a bit more precise we will elaborate the procedure in a little more detail. Therefore we define an abstract search tree: (following the definitions form Hromkovic [2001]):

Let \( M(x) \) denote the set of all feasible solutions for the input instance \( x \). The search tree \( T_{M(x)} \) is then defined as a labeled rooted tree with the properties:

1. Every node \( n \) of \( T_{M(x)} \) is labeled by a set \( S_n \subseteq M(x) \)
2. The root of \( T_{M(x)} \) is labeled by \( M(x) \).
3. The sets associated with the children of a parent node are a partition of the set associated with the parent:
   - If \( n_1, \ldots, n_m \) are children of node \( n \), then \( S_n = \bigcup_{i=1}^{m} S_{n_i} \) and \( S_{n_i} \cap S_{n_j} = \emptyset \) for \( i \neq j \)
4. For every leaf \( u \) of \( T_{M(x)} \), \( |S_u| \leq 1 \) (|\( S_u \)| = 1, leaf contains a feasible solution)

Having these specifications, all feasible solutions can be constructed in a systematic way. The first level consists of as many nodes as the first solution component has values. That means for each possible value of the first solution component a child node is created which has a particular value assigned for the first solution component. All other solution components remain unchanged in the first level. In the second level, the previous process is repeated for each previously created child node. This time for all possible values of the second solution component a node is created. This process is repeated until all solution components are fully instantiated and the search tree is completely constructed. Considering this tree, backtracking means nothing else than a systematic depth-first-search traversal of it.
In practice, the tree is actually not completely constructed because it is usually too big to fit into memory. Instead, it is generated on the fly during the search. To speed up the backtracking search, bounding methods can be used. These are explained in the next subsection.

**Branch-and-Bound**

Branch-and-Bound is a method that makes simple backtracking search more efficient. Early sources can be found in Ignall and Schrage [1965] and Lawler and Wood [1966].

The general idea is to speed up the search process by omitting regions of the search space which cannot contain an optimal solution. To illustrate this, consider the search tree again. Omitting regions of the search space in that sense means just to cut subtrees which cannot contain the optimal solution. To determine these subtrees, a bound on the optimal solution quality must be calculated in advance. This bound\(^1\) tells what the worst quality of an optimal solution could be. Having this bound, it is easy for the search algorithm to omit subtrees. During the search, it simply calculates the quality of partial solutions (inner nodes) and if it encounters a node whose quality value is worse than the worst possible optimal solution, the subtree below that node is not further considered (cut).

The effectiveness of branch-and-bound depends on the size of the pruned subtree. If only small trees are cut, the extra effort of calculating the bound might not pay off. It is also important to note that even with the cleverest method for calculating the bounds, on the cost of the optimal solution, branch-and-bound cannot lead to an algorithm that can solve NP-hard problems in polynomial time. Even if the calculated bounds are very tight, there can always be input instances with exponentially many feasible solutions between optimal cost and calculated bound. Although, for certain problems it is possible to calculate an exact bound which makes it possible to calculate a solution of a certain quality, e.g., 95% in polynomial time.

Within a CP framework, branch-and-bound can be easily implemented by adding additional constraints during the search is performed. Constraint propagation will automatically prune undesirable subtrees.

**2.3 Incomplete Algorithms/Stochastic Search**

**2.3.1 Introduction**

Incomplete algorithms, in contrast to complete systematic algorithms, do not guarantee to find the global optimal solution for a given COP. They are capable of finding solutions that are very close to the optimal solution in a reasonable amount of time. However, it is not guaranteed that they always find such a near optimal solution. Incomplete algorithms can be subdivided into two main categories: constructive algorithms and local search algorithms. Constructive algorithms generate a solution step by step. A prominent and widely used example are greedy construction methods [Hromkovic, 2001]. Greedy methods are very fast, but often only produce poor quality solutions. In comparison, local search algorithms start with an initial solution and try to iteratively replace this solution with a better solution from the local neighbourhood. The drawback of local search techniques is that they might get stuck in poor quality local optima.

In the last 35 years new types of stochastic search algorithms have been developed, they are summed with the term metaheuristics [Blum and Roli, 2003]. They can be regarded as high level search strategies which can be applied to different kinds of Combinatorial

---

\(^1\)upper-bound for minimization problems, lower-bound for maximization problems
Optimization Problems. It has been found that metaheuristics are currently belonging to the best performing approximation techniques for solving COPs [Michalewicz and Fogel, 2000, Blum and Roli, 2003].

2.3.2 Metaheuristics

The term metaheuristic was first used by Glover [1986]. However, there is no commonly accepted definition of this term, the following properties appear to characterize metaheuristics [Blum and Roli, 2003]:

- Metaheuristics are high-level strategies that guide the search process of underlying more problem specific heuristics
- Metaheuristics are incomplete and mostly non-deterministic
- General concepts of metaheuristics can be described independently of a particular problem but need to be instantiated for different problems to be applicable

The main purpose of metaheuristics is to cleverly balance between intensification and diversification. Diversification has the goal to guide the search process into different regions of the search space in order to avoid getting trapped into poor local optima. On the other hand, intensification has the contrary goal to concentrate the search on regions where the solution quality is high. Metaheuristics can basically divided into two categories: trajectory methods and population based methods. In trajectory methods the search process is characterized by a trajectory through the search space. Representatives for trajectory methods are for example Simulated Annealing [Kirkpatrick et al., 1983] and Tabu-Search [Glover, 1986]. Simulated Annealing (SA) is inspired by physical annealing processes. The SA procedure starts with a randomly sampled solution and randomly samples another solution from the neighbourhood of the initial solution. If the newly sampled solution has a better objective value, the initial solution is replaced by it, otherwise the newly sample solution neglected or accepted with probability $T$. If no solution is accepted, the neighbourhood is resampled. The parameter $T$ corresponds to the temperature in real annealing processes, it slowly decreases as the search continues. This has the effect that at the beginning it is rather easy for the algorithm to escape from poor local optima, whereas at the end of the search, it homes in on high quality solutions.

Tabu search is a different trajectory method which is deterministic (in its original form) and not based on any natural phenomenon. The basic idea is to use a short term memory to keep track of recently visited solutions. These solutions are stored in a so called TABU list and are not considered by the search algorithm when it determines the next move. This means that the search algorithm has to visit a new solution in each move, which in consequence allows uphill movements to escape local optima. A short TABU list means that uphill movements are limited and the search concentrates on a relative small area. A longer list in contrast forces the algorithm to explore larger regions as more recently visited solutions are forbidden.

Population based methods differ from trajectory methods as they performed the search process on a set of solutions instead on a single one. The most common methods are Evolutionary Computation (EC) and Ant Colony Optimization (ACO). EC algorithms are based on natural evolutionary processes. They utilize the fact that evolution is capable of evolving organisms which are optimized for their particular environment. EC methods simulate this process as they try to generate solutions for optimization problems which are optimal under a given fitness function. To achieve this EC methods apply recombination
and crossover operators to combine one or more existing solutions to obtain new fitter solutions. Existing EC algorithms can be divided into three basic concepts: Evolutionary Programming [Fogel, 1962, Fogel et al., 1966], Evolutionary Strategies [Rechenberg, 1973] and Genetic Algorithms [Holland, 1975]. EP and ES are mostly used for continuous optimization, whereas GAs play an important role in combinatorial optimization. Although, GAs have been extensively researched they are not covered further here, because of the limited scope of this background section.

**Ant Colony Optimization**

Ant Colony Optimization (ACO) is a metaheuristic which is inspired by the shortest path finding ability of ant colonies [Deneubourg et al., 1990]. It was initially proposed by Dorigo [1992]. Subsequent developments published by the same author can be found in Dorigo and Gambardella [1997], Dorigo and Di Caro [1999]. The following description of ACO is mainly based on Cordon et al. [2002].

Ants live together in colonies and interact with each other in a collaborative fashion. This interaction enables them to solve complex tasks, which would be impossible to solve for single ants. One of these capabilities is the ability of finding shortest paths to food sources around the nest. To perform this complex task ants make use of pheromone trails, each time they walk back to the nest from a food source they deposit pheromone on their path.

Without a pheromone trail in their surrounding ants essentially move around randomly. In case there is a pheromone trail in their neighbourhood they are very likely to follow the trail. While an ant is following a trail it also deposits pheromone at this trail such that the initial pheromone concentration increases. This positive feedback loop enables ants to find shortest paths. A schematic illustration is given in Figure 2.1(a) – 2.1(d).

![Figure 2.1: Shortest path finding behaviour](image-url)
If a number of ants simultaneously start from their nest, the ants finding the closest food source will return back to the nest earlier than the ones which find a more remote sources. In consequence, the branch to the closer food source has already a slightly higher pheromone value associated. The decision of other ants will therefore be more biased to follow the path to the closer source, than to follow the weaker trail. As a result the shorter path will be reinforced and nearly all ants will gather the closest source.

This behaviour has been translated into an algorithmic concept (ACO). ACO algorithms use a pheromone model to simulate the biochemical counterpart. In ACO artificial ants construct solutions by sequentially adding new components to partial solutions. The choice which new component is added is probabilistic and influenced by associated pheromone and heuristic values. The pheromone values can change over time and, strongly simplified, reflect the knowledge about the quality of that particular component in previous complete solutions. The associated heuristic values stay constant while the algorithm is executed. They represent problem specific knowledge and are initialized before the ACO algorithm starts.

ACO algorithms can be applied to all sorts of problems which can be represented as a shortest path finding problem on a weighted graph. A classical example is again the traveling salesman problem.

The first ACO algorithm was proposed by Dorigo in collaboration with Maniezzo and Colomi in 1991 [Dorigo, 1992, Baluja, 1991], it was called Ant System. For the Honours project we use a modified version named ACS [Dorigo and Gambardella, 1997] as the basic ACO algorithm. The pseudo-code for the basic ACO algorithm is given by Algorithm 1 (taken from *Swarm Intelligence* [Eric Bonabeau, 1999]).

**Algorithm 1 Basic ACO algorithm**

/* Initialize pheromone values */
for every edge \( (i, j) \) do
    \( \tau_{ij}(0) = \tau_0 \)
end for
for \( k = 1 \) to \( m \) do
    Place ant \( k \) on a randomly chosen city
end for
/* Main Loop */
for \( t = 1 \) to \( t_{\text{max}} \) do
    for \( k = 1 \) to \( m \) do
        Build a solution \( S_k(t) \) by applying \( n - 1 \) times a probabilistic construction/modification rule where choices are a function of a pheromone trail \( \tau \) and of an heuristic desirability \( \eta \)
    end for
    for \( k = 1 \) to \( m \) do
        Compute the cost \( C_k(t) \) of the solution \( S_k(t) \) built by ant \( k \)
    end for
    if an improved solution is found then
        update best solution found
    end if
    for every edge \( (i, j) \) do
        Update pheromone trails by applying a pheromone trail update rule
    end for
end for
Print the best solution

As mentioned before, ACO algorithms can be applied to a huge number of different combinatorial optimization problems for example the TSP, the quadratic assignment problem (QAP) [Maniezzo et al., 1994], job-shop scheduling [Colori et al., 1994], vehicle routing [Colori et al., 1999] or sequential ordering [Gambardella and Dorigo, 2000].
2.3.3 Constraint Handling

Most real world optimization problems are subject to constraints. If we consider the above TSP example again it is obvious that in reality such a problem never occurs in that unconstrained way. It is more realistic that for example the traveling salesman has to be in a certain city at a specific time, or some cities have to be visited after each other. The fact that real problems almost always appear in connection with constraints rises the question: How to handle these constraints algorithmically?

In the past decades various constraint handling methods were developed. A survey of techniques used in evolutionary computation is given by Michalewicz [1995b]. The simplest way to handle constraints is probably to use death penalties [Schwefel, 1981]. This method just ignores all solutions which violate a constraint. The respective search algorithm just generates new solutions until a feasible one is generated. Using death penalties might be appropriate for some problems but is likely to fail in situations when the search algorithm has to leave a feasible region of the search space to find the actual optimal solution in a non adjacent region. In such a case infeasible solutions would act as bridging solutions between different feasible regions. Michalewicz [1995a] has shown that in this case approaches which adjust penalization dependent on the distance to the feasible region perform clearly superior to death penalization.

The death penalty approach is also impracticable, if it is itself very hard to find an initial feasible solution. The search algorithm would not have a solution to start with and therefore no chance to find feasible regions.

The next two sections describe how constraints can be handled different from death penalization. At first, the general concept of penalty functions is introduced. Secondly, stochastic ranking, a more recent, technique is explained.

Penalty Methods

Penalty Functions

Penalty functions are the most common approach to handle constraints in evolutionary computation [Joines and Houck, 1994, Michalewicz et al., 1996, Smith and Coit, 1997]. Let us consider the following general non-linear programming problem:

\[
\text{minimize } f(x) \\
x \in S \cap F, S \subseteq \mathbb{R}^n
\]

where \( S \) is the \( n \)-dim search space and \( F \) the feasible region is defined by:

\[
F = \{ x \in \mathbb{R}^n | g_j(x) \leq 0 \ \forall j \in \{1 \ldots m\} \}
\]

where \( g_j(x) \) represent \( m \) constraints on \( x \).

The general idea behind using penalty functions is to transfer the above constraint problem into an unconstraint one. This transformation is simply achieved by adding or subtracting a value to/from the objective function \( f(x) \) proportional to the amount a given solution violates constraints. The transformed objective function then becomes \( \Psi(x) = f(x) + \phi(g_i(x)) \ i = 1 \ldots m \), where \( \phi(.) \) is the penalty function that penalizes constraint violations. An often used penalty function is the quadratic loss function.
\[
\phi(g_i(x)) = \sum_{j=1}^{m} \max\{0, g_j(x)\}^2
\]

A decrease in value of this function means that the constraint violation decreases and a solution tends to approach a feasible region. It therefore guides the search to feasible regions.

A big difficulty that comes along with the usage of penalization terms is the choice of the right amount of penalization. In a scenario where the penalization is chosen to high, the search algorithm will have problems to leave a feasible region. It will be impossible to use infeasible space as bridging space between feasible islands. The same is true, if the actual global optimal solution is located near the border to an infeasible region. In this case a high penalization will push the search algorithm far back into the feasible space each time it crosses the border to the infeasible region.

In contrast, if constraint violation is penalized to low, the performance of the search algorithm will also degrade. In this case the algorithm will waste a lot of search time in infeasible regions because the objective function dominates its search. That means the amount of constraint violation plays only a minor role.

For an overview of penalty functions and other constraint handling techniques in evolutionary computation we refer to Michalewicz [1995b]. The next section describes a constraint handling technique that is based on a ranking procedure which avoids the difficult problem of choosing the right amount of penalization.

**Stochastic Ranking**

Stochastic ranking (SR) is a rather new method which was originally developed by Runarsson and Yao [2000] as constraint handling technique for evolutionary optimization. The usual way of handling constraints in evolutionary optimization is to rank the solutions (individuals) according to a fitness function. Solutions with a better fitness get a higher change to survive and be present in the next evolutionary iteration than solutions with a smaller fitness. The value of the fitness function is determined by the quality of the solutions in terms of objective value and constraint violation. As indicated above, finding an optimal fitness function can be very hard because it can be an optimization problem itself to find the optimal balance between objective and penalty function.

SR instead uses a different approach to rank solutions. It requires one parameter \(p_f\) to balances between objective and penalty function. SR uses a ‘stochastic’ version of bubble sort to rank solutions. This procedure stochastically balances between objective and penalty function. The pseudo code for the procedure (for a minimization scenario) is given below:

If a solution is feasible it is always ranked according to its objective value. If a solution is infeasible it is ranked according to its degree of constraint violation or with a probability \(p_f\) according to its objective value. Given that procedure it becomes clear that \(p_f\) is the only parameter which must be adjusted to balance between objective and penalty function. If \(p_f\) is set to 1, all solutions would be ranked only according to their objective values regardless how infeasible they are. This is a under penalization scenario. On the other hand, if \(p_f\) is set to 0 all infeasible solutions will be ranked only according to their penalization values, which is in this case over penalization.

To find and appropriate value for \(p_f\) the particular optimization algorithm must be run several times with different values for \(p_f\). In the original paper by Runarsson and Yao [2000] it has been found that a \(p_f\) value of 0.5 appeared to be optimal for the given benchmark set of continuous optimization problems. Furthermore, the authors report
for $i \leftarrow 0$ to $n$ do
  for $j \leftarrow 1$ to $\lambda - 1$ do
    if $[(\text{feasible}(I_j) \text{ AND feasible}(I_{j+1})) \text{ OR } \text{random()} < pf)]$ then
      if $[\text{objectiveValue}(I_j) > \text{objectiveValue}(I_{j+1})]$ then
        swap($I_j, I_{j+1}$)
      end if
    else
      if $[\text{penaltyValue}(I_j) > \text{penaltyValue}(I_{j+1})]$ then
        swap($I_j, I_{j+1}$)
      end if
    end if
  end for
end for

Figure 2.2: Stochastic Ranking procedure, $I_j$ represent a solutions/individual with index $j$

that SR applied to the same set of 13 benchmark problems performed superior to classical
dynamic penalty methods in 10 cases.

Recently SR was ported to handle constraints within an ACO framework [Meyer,
2005b]. The obtained solution quality was very good as well, for weakly and medium
constrained problems it was able to compete with a computationally far more expensive
constraint programming technique.

2.4 Constraint Programming and Metaheuristic Hybrids

In the previous sections two different approaches to solve constraint combinatorial opti-
mization problems were introduced.

The systematic approaches like constraint programming with simple backtracking
search are guaranteed to find the optimal solution for a problem (given a solution ex-
ists), but it may take an impractically large amount of time because the whole search
space is searched exhaustively. Nevertheless, if the given problem is associated with a
large number of hard constraints, constraint programming techniques can make effective
use of constraint propagation to prune the search space and therefore solve the given
problem in a reasonable amount of time.

Stochastic search methods, on the other hand, have their strength in finding nearly
optimal solutions for a problem within a short amount of time, but their performance often
degrades when the given problems are constrained. Especially when hard constraints are
present, stochastic metaheuristics that use penalty methods to handle constraints tend to
perform poorly.

These observations led to the idea of hybridizing metaheuristics and constraint pro-
gramming with the aim to obtain an algorithm that combines the strength of both tech-
niques. Focacci et al. [2003] present a number of successful integrations of CP techniques
in metaheuristics. They outline different ways of integrating CP into local search methods
as well as into constructive global search methods.

Below we describe two recent approaches of such integrations. The first one, CPACS
[Meyer and Ernst, 2004], integrates CP and ACO to solve single machine job schedul-
ing problems. The second, combines CP with local search in a coupling with different
metaheuristics to solve Vehicle Routing Problems.
2.4.1 CPACS

CPACS [Meyer and Ernst, 2004] is a hybrid algorithm which combines an ACO algorithm (Ant Colony System [Dorigo and Gambardella, 1997]) with Constraint Programming (CP). The basic idea of CPACS is to use constraint programming techniques to assist the ants while they are constructing tours. When an ant in the normal ACO scheme has to decide which city or job to visit or schedule next, it considers all cities/jobs which have not been scheduled before, regardless of constraints that might be violated in consequence. With the use of constraint programming and associated constraint propagation techniques this process can be improved such that the domain of possible cities/jobs that can be visited/scheduled next is automatically restricted by the underlying constraint program. Only cities/jobs which not lead to a constraint violation if they are visited/scheduled next can therefore be considered by the tour constructing ant. Due to the often NP-hard nature of the given problems constraint propagation cannot be complete, it might therefore happen that ants still fail if they construct tours.

However, CPACS was found to perform very well for problems of intermediate tightness. That means for problems where the search space is on the one hand to large to be searched with plain CP techniques but on the other hand also too fragmented that it becomes complicated for ACO to search it effectively. Besides these positive results the paper poses some open research questions which should be addressed. One of them is the question when it pays off to use constraint propagation in terms of run-time. As constraint propagation can be computational very expensive dependent on the triggered propagation steps, it is not always clear if investing the same amount of time for a broader plain ACO search would lead to similar or better solutions.

2.4.2 Local Search and Constraint Programming

Another example for the successful hybridization of constraint programming and metaheuristics is given in Backer et al. [2000]. The paper describes the application of local search in combination with constraint programming (CP) to solve vehicle routing problems. The local search is performed by moving through the search space by applying different move operators which modify small parts of constructed solutions. To test the validity of a potential move a CP system is used. If the CP solver tells that a move would reduce the domain of a constraint variable to empty this move would become invalid and not considered during the local search procedure. In order to avoid full constraint checking for each potential move, potential moves are pre-filtered. Only moves that meet specific criteria, for example cost reducing, are considered to be further checked with the CP system.

To bypass the problem of plain local search getting trapped in local optima the search procedure is coupled with different metaheuristics. The authors investigated couplings with different metaheuristics, i.a. tabu-search, guided local search (GLS) and a guided tabu search (combination of tabu search and GLS). Examinations of these methods on a set of standard benchmark problems revealed their superiority to existing approaches. For some problems the hybrid methods were able to find several new best solutions. One has to notice that the obtained results might have been outperformed by other approaches since the paper was published, but they still reflect that a combination of constraint programming and metaheuristics is a reasonable approach which should be studied further.

In terms of runtime the authors mention that the metaheuristic approach performs almost the same number of moves per second as a plain local search strategy. Unfortunately, they do not discuss the runtime consumption of the CP system. It might have been worthwhile to investigate the payoff between using a sophisticated CP system and
alternative techniques to handle constraints, as it is possible that other approaches lead to the same solution quality when they invest the same amount of time.

Constraint programming was also used in combination with a genetic algorithm by Deris et al. [1999] to solve university timetabling problems.
Chapter 3

Stochastic Ranking and ACS

3.1 Introduction

Stochastic Ranking (SR) is a constraint handling technique which was originally introduced in the context of evolutionary optimization [Runarsson and Yao, 2000]. It attempts to simplify the rather complicated process of finding the right balance between objective and penalty function by applying a stochastic sorting procedure to rank different solutions. An advantage over other constraint handling methods is given by the fact that it requires only one parameter to be set. For a more thorough description of SR refer to the Stochastic Ranking part of Section 2.3.3 in the Background chapter.

More recently SR was combined with Ant Colony Systems (ACSsr) [Meyer, 2005b] to tackle single machine job scheduling problems. SR was used to rank all solutions (job assignments) that were obtained in one ACS iteration. Only the solution that ranked best was subsequently rewarded by depositing pheromone on the corresponding path. The results that could be obtained by applying this combination to a set of test problems were quite promising [Meyer, 2005b], especially in comparison to results that were gained by applying a computationally more expensive constraint programming ACS hybrid (CPACS) to the same problem set [Meyer and Ernst, 2004]. ACSsr outperformed CPACS in terms of solution quality for weakly and moderately constrained problems. This is quite remarkable, because the computational complexity of ACSsr is far smaller than the one for CPACS, as it does not employ a sophisticated constraint solver. These interesting results gave rise to a further investigation of the actual SR mechanism. As SR was beforehand only studied in the context of Evolutionary Optimization. There a top ranking fraction of SR ranked solutions were used to generate new solutions. On the other hand, in ACSsr only the top ranking solution receives a pheromone reward. It was therefore not clear if the ranking process could be modeled by a procedure which just rewards certain solutions based on a simple probability distribution. In fact, external empirical studies with simulated data [Meyer, 2005a] give evidence for this conjecture. In the mentioned studies SR was applied to a number of solutions with different objective\textsuperscript{1} and constraint violation\textsuperscript{2} values. Different values for the parameter $pf$ were tested. For $pf = 0.7$ it emerged that the probability for the best makespan solution to bubble up to the top rank was equal to the probability of the best infeasible\textsuperscript{3} solution to come top. This particular parameter setting was also used when ACSsr was applied to real data [Meyer, 2005b]. Hence, the question arose if the SR procedure behaves on real problem data in a similar way as it did on the simulated data. If this conjecture would be true, SR could be replaced by a simple

\textsuperscript{1}total makespan in context of job scheduling
\textsuperscript{2}total tardiness in context of job scheduling
\textsuperscript{3}smallest tardiness

15
‘coin-tossing’ procedure that rewards either the best makespan solution or the best infeasible solution with a probability of 50%. To examine this question we implemented such a ‘coin-tossing’ procedure and applied it to the problem set also used in [Meyer, 2005b].

The following Methods section describes the implemented algorithm and the testing procedure. It is followed by the Results section that presents the outcomes of the comparison. Finally, the findings are evaluated in the Discussion section.

### 3.2 Methods

To investigate the difference between the two implementations we run ACS in combination with SR and in a variation where the rewarding depends on a single decision. In the following we term the later version ACScoin.

We use the standard ACS algorithm as basis and replace its pheromone rewarding scheme by the Stochastic Ranking procedure. SR changes the rewarding scheme in a way that the solution that comes top after ranking is rewarded instead of always rewarding the globally best solution. For the ranking procedure all solutions created in one iterations as well as the currently best feasible solution are considered. The pseudocode for the described procedure is given in Algorithm 2.

ACScoin used the same ACS version as basis algorithm but replaces the ranking and rewarding by a probabilistic decision. The decision is made between rewarding either the solution which is best in terms of objective (best makespan) or the solution which is best in terms of constraint violation (smallest tardiness). Both choices have a probability of 50%. This models the effect that was observed for the top ranking solution when $pf = 0.7$ was used.

The sr-rank() procedure and its associated rewarding scheme in Algorithm 2 is in ACScoin replaced by a probabilistic if-then-else construct that models the 50:50 chance. It is given in Algorithm 3.

### 3.2.1 Testing Procedure

To assess if the ACScoin version behaves in a similar way as the ACSsr when applied to real data, both were applied to a set of single machine job scheduling problems (Appendix A). The number of iterations (line x) were set to 5000 and the number of ants per iteration (line y) were set to 10. For both algorithms 10 independent runs were performed on each problem.

### 3.3 Results

ACSsr and ACScoin were both applied to the same set of test instances (Appendix A). Each algorithm was run 10 times on each problem. The results are displayed in Table 3.1. The column Best represents the smallest makespan that was found during the 10 independent runs of the algorithms. Avg is the average makespan for a particular problem over 10 runs. Error Rate gives the percentage of runs were the respective algorithm failed to construct a feasible solution for the given problem.

The results show that the quality of solutions found for the job scheduling problems is quite similar for both tested versions. Differences in best makespan found and average makespan are rather small and not significant considering the stochastic nature of the algorithms.

A more apparent difference is given in terms of error rate. Regarding to this measure ACScoin performs for particular problems (W30.1, RBG16.a, RBG27.a.15, BR17.a.10, BR17.a.17) clearly worse than ACSsr. Examining the properties of these problems reveals
Algorithm 2 ACSsr: ACS in Combination with Stochastic Ranking

1: \( \forall i,j: \tau_{i,j} = \tau_0 \) // initialize pheromone matrix
2: \( \forall i,j: \eta_{i,j} = d\text{ue}(j) \) // initialize heuristic values
3: \( j_{00} = 0 \) // artificial start job 0
4: \( \forall i,j: \tau_{i,j} = \tau_0 \land \eta_{i,j} = 1 \)
5: \( T^{ib} = \text{nil}; T^{ifb} = \text{nil} \) // initialize global best feasible and infeasible
6: for \( t = 1 \) to \( \text{max\_iterations} \) do
7:   for \( k = 1 \) to \( \text{number\_of\_ants} \) do
8:     \( T_k = \text{nil} \) // initialize tour \( k \) as empty
9:   mark all jobs as unscheduled by ant \( k \)
10: for \( n = 1 \) to \( \text{number\_of\_jobs} \) do
11:     \( C = \text{set of jobs not yet scheduled by ant } k \)
12:     \( i = \text{job}_{n-1} \)
13:     if \( \text{random}() > p \) then
14:       choose next job \( j \in C \) to be scheduled by ant \( k \) with probability
15:       \( p_j = \frac{\tau_{i,j}^\alpha \cdot \eta_{i,j}^\beta}{\sum_{j \in C} \tau_{i,j}^\alpha \cdot \eta_{i,j}^\beta} \)
16:     else
17:       \( j = \text{arg max}_{j \in C} (\tau_{i,j}^\alpha \cdot \eta_{i,j}^\beta) \)
18:     end if
19:     \( T_k = \text{append}(T_k, (i, j)) \)
20:   end for
21:   if \( \text{feasible}(T_k) \) then
22:     if \( \text{makespan}(T_k) < \text{makespan}(T^{fb}) \) then
23:       \( T^{fb} = T_k \)
24:     end if
25:   else
26:     if \( \text{makespan}(T_k) < \text{makespan}(T^{ifb}) \) then
27:       \( T^{ifb} = T_k \)
28:     end if
29:   end if
30: end for
31: \( x_1 = T^{fb} \)
32: \( x_2 = T^{ifb} \)
33: \( \forall i \in \{1 \ldots \text{number\\_of\\_ants} \}: x_{i+2} = T^a \)
34: sr-rank() // with \( f(\cdot) = \text{makespan} \) and \( g(\cdot) = \text{tardiness} \)
35: \( T^{srb} = x_1 \) // stochastically best ranked tour
36: \( \forall i,j \in T^{srb}: \tau_{i,j} = (1 - \rho) \tau_{i,j} + Q \cdot \text{makespan}(T^{srb})^{-1} \) // evaporate and reward
37: end for

that they are very tight, which means finding feasible solutions for them itself is hard, regardless of finding an optimal solution. The higher error rate of ACScoin for these problems indicates that the algorithm is occasionally not able to find feasible solutions at all, whereas ACSsr apparently always finds feasible solutions for all scheduling problems in the test set, except for RGB27.a.27 where both algorithms fail completely.

This result consequently shows that the SR procedure as it is integrated into ACS is more complicated than the simulated data suggests. It cannot simply be replaced by a procedure that either rewards the best feasible solution or the solution with the smallest constraint violation with a probability of 50%, as it was previously suggested by the simulation results [Meyer, 2005a].

An inspection of the used simulation conditions revealed that all simulation data sets contained a fraction of feasible solutions, they did not contain of a set comprising only infeasible solutions [Meyer, 2005a]. That means in consequence that the simulations which left to the ‘coin-tossing’ conjecture did not consider scenarios in which all solutions are infeasible. This is a shortage as it does not account for very tight real world problems were the ant generated solution set only contains infeasible solutions.
Algorithm 3 ACScoin: Replacement for ACSsr lines 35-37

if makespan($T^{fb}$) < makespan($T^{ifb}$) then
  $T^{cb} = T^{fb}$
else
  if random() > 0.5 then
    $T^{cb} = T^{ifb}$
  else
    $T^{cb} = T^{fb}$
  end if
end if

$\forall_{i,j} \in T^{cb} : \tau_{i,j} = (1 - \rho)\tau_{i,j} + Q \cdot \text{makespan}(T^{cb})^{-1}$ // evaporate and reward

<table>
<thead>
<tr>
<th>Problem</th>
<th>ACScoin</th>
<th>ACSsr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Avg</td>
</tr>
<tr>
<td>W8.1</td>
<td>8321.0</td>
<td>8321.5</td>
</tr>
<tr>
<td>W8.2</td>
<td>5818.0</td>
<td>5818.0</td>
</tr>
<tr>
<td>W8.3</td>
<td>4245.0</td>
<td>4245.0</td>
</tr>
<tr>
<td>W20.1</td>
<td>8504.0</td>
<td>8543.5</td>
</tr>
<tr>
<td>W20.2</td>
<td>5082.0</td>
<td>5107.0</td>
</tr>
<tr>
<td>W20.3</td>
<td>4312.0</td>
<td>4354.5</td>
</tr>
<tr>
<td>W30.1</td>
<td>7977.0</td>
<td>8087.56</td>
</tr>
<tr>
<td>W30.2</td>
<td>4642.0</td>
<td>4696.5</td>
</tr>
<tr>
<td>W30.3</td>
<td>4193.0</td>
<td>4297.0</td>
</tr>
<tr>
<td>RBG10.a</td>
<td>3840.0</td>
<td>3840.0</td>
</tr>
<tr>
<td>RBG16.a</td>
<td>2596.0</td>
<td>2596.0</td>
</tr>
<tr>
<td>RBG16.b</td>
<td>2094.0</td>
<td>2100.1</td>
</tr>
<tr>
<td>RBG21.9</td>
<td>4487.0</td>
<td>4502.1</td>
</tr>
<tr>
<td>RBG27.a.3</td>
<td>1667.0</td>
<td>1684.4</td>
</tr>
<tr>
<td>RBG27.a.15</td>
<td>1441.0</td>
<td>1442.0</td>
</tr>
<tr>
<td>RBG27.a.27</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>BR17.a.3</td>
<td>1471.0</td>
<td>1510.0</td>
</tr>
<tr>
<td>BR17.a.10</td>
<td>1398.0</td>
<td>1398.0</td>
</tr>
<tr>
<td>BR17.a.17</td>
<td>1057.0</td>
<td>1057.0</td>
</tr>
</tbody>
</table>

Table 3.1: Performance Results for ACScoin and ACSsr

Analyzing the distribution of solutions in the first rank after rerunning the simulations with only infeasible (tardiness > 0) solutions showed that the top ranking solution was dependent on the average degree of constraint violation in the solution set. Therefore it is not clear if a simple procedure exists that models that kind of distribution. Examining this question is out of the scope of this thesis.

It has been shown that a simple replacement of the SR procedure by a probabilistic ‘coin-tossing’ rewarding scheme was not able to achieve similar performance results. We can therefore conclude that the SR procedure integrated in ACS is justified and cannot easily be replaced by a rather simple mechanism.

High failure rates of 100% for problem RBG27.a.27 indicate that this problem is apparently so highly constrained such that neither of the two algorithms is able to generate a feasible solution. Such heavily constrained problems can be better tackled by a search algorithm which utilizes constraint programming techniques to generate feasible solutions.
Chapter 4

Constraint Programming and ACS

4.1 Introduction

The previous chapter introduced one particular way (Stochastic Ranking) of handling constraint violations within an Ant Colony Optimization framework. In addition to Stochastic Ranking there are a number of other techniques which allow to handle constraints within meta heuristics, for example, penalty-based techniques or repair methods. All these techniques have in common that they handle constraint violations in a soft way. They allow infeasible solutions to be generated and used while the search process is carried out. In contrast to soft handling techniques there exist hard constraint handling techniques. The fundamental difference in handling constraints in a hard way is not to utilize any infeasible solution during the search. This can either be accomplished by completely ignoring all infeasible solutions or by augmenting the search procedure with a form of lookahead in a way that it only generates feasible solutions. A way to achieve the later is to make use of Constraint Programming. Refer to Section 2.2.2 in the Background chapter for a short introduction to CP.

CP performs quite well when it is used to solve tightly constrained optimization problems. By making use of constraint propagation CP is able to reduce the search space of such problems significantly, such that the remaining search for the optimal solution comprises only a comparatively small set. Unfortunately, the performance of CP degrades for optimization problems that are moderately or weakly constrained. The search space reductions achieved for such problems by CP are is rather small and the space of remaining feasible solutions is comparatively large. In consequence the search component that is used in combination with CP becomes more important. Simply enumerating and evaluating all remaining feasible solutions via normal backtracking search is inappropriate for such cases. The usage of clever search strategies becomes necessary.

Ant Colony Optimization (introduced in Section 2.3.2) is an example for such a strategy. Apart from other meta heuristics ACO has been found to perform competitively in solving combinatorial optimization problems. Unfortunately, its performance degrades when it is applied to constraint problems. In its original form, ACO just ignores infeasible solutions and attempts to regenerate feasible solutions by chance. For problems that are tightly constraint, i.e, have a small space of feasible solutions, this approach is apparently not suitable. However, as outlined above it exhibits a good search performance on unconstrained or weakly constrained problems. This together with a relatively straightforward way of integrating CP makes ACO a good candidate for a hybrid algorithm that combines the benefits of constraint programming with the advantages of clever search algorithms.

CPACS [Meyer and Ernst, 2004] is one example for such a hybrid algorithm that combines Constraint Programming (CP) and Ant Colony Optimization (ACO). CP is integrated in a way that it supports the virtual ants with constructing feasible solutions.
When an ant decides which city to visit next, CP attempts to automatically rule out assignments which would later in the construction process lead to a constraint violation and therefore make the constructed tour infeasible. An example for that process is given in Figure 4.1.

![Search Tree of an CP Assisted Ant](image)

**Figure 4.1:** Search Tree of an CP Assisted Ant - Numbers at edges represent traveling time between cities, pair of numbers at nodes represent time windows in which the respective cities have to be visited (#release, #deadline), dashed lines mark possible paths ruled out by constraint propagation.

The ant has visited city A and has now to decide which city to visit next. Without utilising CP there would be three possibilities to choose from, but with a CP model working in the background, that is aware of all problem constraints, city D is ruled out because by using edge \((A, D)\) the used time would accumulate to 40 and therefore violate the due date constraint of city D.

and only two choices remain. City A has already been visited and it has has a time window of \((10, 50)\), which means that the time that already has past is 10. Would the ant decide to visit D next it would have to spend 30 time units to get there. The time would therefore add up to 40, which would violate the deadline constraint of city D. A similar argument applies for the exclusion of city B after visiting A-C and city B after visiting A-C-D.

Using CP to prevent the construction of infeasible solutions means that the feasible search space can be explored more widely in comparison to a standard ACS algorithm. Considering for both algorithms the same number of tour constructions, the one with CP support will construct more feasible solutions and therefore explore the search space more exhaustively, which results in a higher chance of finding a high quality solution.

However, using CP cannot always guarantee that only feasible solutions are constructed. It might happen that ants run into a dead end and fail to construct a feasible
tour. The likelihood for such an event is associated with the amount of lookahead that is provided by the CP system. Let us consider the example in Figure 4.1 again. In this case the lookahead is rather small. Scheduling city $C$ next would lead into a dead end. With more constraint propagation city $C$ would be ruled out by the CP system of the set of possible cities. In the given example the domain of the second job slot is $d(job_2) = \{B, C\}$. The constraint checking of the solver checks for all contained values if an assignment of them would reduce the domain of other decision variables to zero. If this is the case, this particular value is removed. In our example it would check for $C$ that the domain of job slot 3 would be reduced to $d(job_3) = \{D\}$ from initially $\{B, C, D\}$. $C$ is removed because it has been scheduled already and $B$ is removed because a scheduling of $C$ after $A$ would increase the spent time to 20. Using the edge to $B$ would cost another 15 and therefore violate the the time window of $B$. Having reduced the domain of $job_3$ to $\{D\}$, its now the question how much propagation is carried out. A strong propagation solver would check the effect a scheduling of $D$ at the third position would have. It would notice that the domain of $job_4$ becomes empty because it contained only $B$, but $B$ cannot be scheduled because its time window would be violated if scheduled after scheduling $A, C$ and $D$. The information that the domain of $job_4$ becomes empty would propagate back to $job_3$ and remove $D$ from its domain. This in turn would narrow down the domain of $job_3$ to be empty and propagate to $job_2$ and force the removal $C$ such that only $B$ can be chosen to be visited next. Hence, the strength of constraint propagation determines how often an ant will run into dead ends and die. The more constraint propagation the more lookahead is provided. Which more lookahead in turn the search efficiency increases as more promising regions are explored considering the same number of used ants.

It is important to mention that a constraint solver cannot be complete, meaning never all effects of a variable assignment can be considered. This limitation is due to the nature of the problems which are usually attempt to solve with a CP system. These problems are usually NP-hard, therefore a CP solver in order to be usable must be incomplete.

As described, the lookahead provided by the CP system is crucial for the effectiveness of the search. In that context, one task of this thesis is to improve on this lookahead by strengthening the CP solver. Besides the general benefits of improving the lookahead strengthening the solver became necessary because the utilized constraint programming library ILOG JSolver appeared to be rather weak in terms of constraint propagation (Appendix B.1).

To achieve this we implement a domain filtering algorithm that removes values from the domain of unassigned jobs that cannot improve on the best so far found solution. As basis for this filtering a linear relaxation of the original problem is used. Solving this relaxation provides a lower bound on the solution of the original problem. The obtained bound together with additional information provided by the linear relaxation allows us to exclude additional values from variable domains\(^1\) which might have not been pruned by the CP system. In order to measure the effect such an additional pruning has on the effectiveness of the search, the performance of CPACS with additional domain pruning is compared to the standard version. We are investigating the effect a stronger solver has on the quality of the found solutions and the speed in terms of labeling steps that are on average needed to obtain a certain solution.

A positive side effect of solving the employed linear relaxation is that it provides us information about how ‘desirable’ certain variable assignments are. We investigate the effect of using this extra information to replace the static heuristic values that, together with the pheromone concentration, normally determine the probability of an edge to be selected by an ant. To assess the effect of the dynamic guidance its performance on the

---

\(^1\)In relation to the example (Figure 4.1, one could speak of removing cities from the list of potential next cities)
standard test set (Appendix A) is compared to a version that calculates the edge selection probability in the standard way.

The last part of this chapter is concerned with examining the effect of combining hard- and soft constraint handling techniques in one algorithm. So far we have seen ACSsr that handles constraint violations in a soft way and CPACS which does it in a hard fashion. Both variants have complementary strength for different types of problems. ACSsr appears to perform quite well for problems of moderate and medium tightness, whereas its error rate increases for very tight problems. CPACS, on the other hand, exhibits good results for medium and hard constrained problems but is outperformed for weakly constrained problems. In this context we investigate if a coupling of both approaches in one algorithm is worthwhile, i.e., the performance of a combined version is better than the respective best performing stand alone algorithm. To examine this question the coupled version is applied to the standard set of various tight problems to empirically compare its performance against the stand alone versions.

The following sections of this chapter are structured as follows. At first, the employed methods are described. Secondly, results that were obtained after applying the developed algorithms to a set of single machine job scheduling problems are presented. And the last section discusses our findings and relates them to each other.
4.2 Methods

4.2.1 Hybridization of CP and ACS

ACS allows us a relatively straightforward integration of constraint programming. The fundamental idea is to make the set of schedulable jobs a constraint variable (denoted by $C$ in Algorithm 4 line 16). The domain of this variable is represented by the set of jobs which are allowed to be visited. The elements in this set are now also determined by the constraint system rather than only by the set of already scheduled jobs.

It is easy to see that this requires a constraint variable for each sequence position, we denote the corresponding variable by $job_i$ (where $i = 1 \ldots \text{number of jobs}$). The pseudocode of the basic CPACS algorithm is given in Listing 4. The domain of $job_i$ should only contain jobs that lead to a feasible tour if chosen. However, as indicated above the used solver and the utilized constraint model (detailed below) might not be strong enough to remove all constraint violating jobs. Therefore it is possible that an assignment $job_i = j$ leads to a constraint violation, but is only discovered once $job_i = j$ is posted to the solver. In order to prevent constructing infeasible tours in these cases we apply a single level backtracking procedure (Algorithm 4 lines 15–26). If it is discovered that an assignment $job_i = j$ fails, the constraint $job_i \neq j$ is posted to the solver to remove $j$ from the domain of $job_i$. The ant attempts then to probabilistically select another job from the domain of $job_i$. This process is repeated until either an assignment is successful or the domain of $job_i$ becomes empty. In the later case the ant ‘dies’ and its so far constructed tour is forgotten and not rewarded with pheromone. To further increase the amount propagation we add a less than constraint on the end time of the last job each time a new best solution is found. This allows the solver to rule out assignments that cannot improve the solution found so far.

Constraint Model

The following model is used to capture the properties of the job scheduling problem in the CP system. It is equivalent to the model used by Meyer and Ernst [2004]. As aforementioned we use decision variables $job_i$ for each sequence position to model jobs that can be assigned to that position, where $job_i = i$ if the $n$-th job in a tour is $i$. The initial domain of these variables is given by $\forall i : job_i \in \{1, \ldots, \text{number of jobs}\}$. We constraint the job variables to take all different values by adding the high level constraint:

$$\text{allDifferent}(job_i)$$

To model the attributes of the problem we introduce three further constraint variables: $start_n$, the start time of the job in sequence position $n$, $end_n$ the respective end time and $setup_n$, the setup time needed to get from job in sequence position $n - 1$ to the job in sequence position $n$. We also make use of three auxiliary variables $duration_n$, $release_n$ and $due_n$ to include the corresponding problem data into the model. Having defined the variables we add the following constraints to capture the relationship between them:

$$\forall n : end_n = start_n + duration_n$$

End time of job in slot $n$ is determined by the sum of its start time and duration.

$$\forall n : start_n \geq release_n$$

The start time of a job must always be greater or equal than/to its release date.

$$\forall n : end_n \geq due_n$$
A job must finish before or when it is due.
\[ \forall n > 1 : \text{start}_n \geq \text{end}_{n-1} + \text{setup}_n \]
Start time of a job must be greater or equal to the finishing time of the previous job plus the time it needs to set it up.

The domains of the auxiliary variables are given by:
\[ \forall n > 1 : \text{setup}_n \in \{ \text{setup}_{value}_{i,j} | i, j \in \{1 \ldots n\} \} \]
\[ \forall n : \text{duration}_n \in \{ \text{duration}_{value}_j | j = 1 \ldots n \} \]
\[ \forall n : \text{release}_n \in \{ \text{release}_{value}_j | j = 1 \ldots n \} \]
\[ \forall n : \text{due}_n \in \{ \text{due}_{j} | j = 1 \ldots n \} \]
where <entity>_value represents the given data associated with the problem.

To bind the problem data to the auxiliary variables as soon as a job is assigned to a sequence position we use *imply* constraints. This type of constraint only propagates once its precondition is fulfilled.

\[ \forall i > 1, l, m : (\text{job}_{i-1} = l \land \text{job}_i = m) \Rightarrow \text{setup}_i = \text{setup}_{value}_{l,m} \]
Setup time of job in slot \( i \) gets the setup time of job \( l \) and \( m \) assigned, if the previous job is \( l \) and the job in slot \( i \) is \( m \).

\[ \forall n, j : (\text{job}_n = j) \Rightarrow \text{duration}_n = \text{duration}_{value}_j \]
If job in slot \( n \) is \( j \), \( \text{duration}_n \) is the actual duration of job \( j \).

\[ \forall n, j : (\text{job}_n = j) \Rightarrow \text{release}_n = \text{release}_{value}_j \]
If job in slot \( n \) is \( j \), \( \text{release}_n \) is the actual release date of job \( j \).

\[ \forall n, j : (\text{job}_n = j) \Rightarrow \text{due}_n = \text{due}_{value}_j \]
If job in slot \( n \) is \( j \), \( \text{due}_n \) is the actual due date of job \( j \).

\[ \forall n, j : (\text{job}_n = j) \Rightarrow \text{start}_n = \max(\text{release}_{value}_j, \text{end}_{n-1} + \text{setup}_n) \]
If job in slot \( n \) is \( j \), its start time is the maximum of its release time and the end time of the previous job plus the needed setup time.

\[ \forall i, j : (\text{end}_i > \text{due}_{value}_j) \Rightarrow \text{job}_i \neq j \]
A job that has a due date bigger than the assigned end time of a slot, this job cannot be scheduled at this position.

To implement the described CPACS algorithm we used Java in combination with the commercial constraint library ILOG JSolver [Solver, 2003].
Algorithm 4 Basic CPACS – combination of CP and ACS

1: initialize solver; post initial constraints; \( s_0 = \text{solver.state()} \)
2: \( \forall i,j : \tau_{i,j} = \tau_0 \) // initialize pheromone matrix
3: \( \forall i,j : \eta_{i,j} = \text{due}(j) \) // initialize heuristic values
4: \( \text{job}_0 = 0 \)
5: \( \forall i,j : \tau_{i,j} = \tau_0 \land \eta_{i,j} = 1 \) // artificial start job 0
6: \( T^{gb} = \text{nil}; l_{gb} = \infty \); // initialize global best
7: for \( t = 1 \) to \( \text{max.iterations} \) do
8: restore initial solver state \( s_0 \)
9: for \( k = 1 \) to \( \text{number.of.ants} \) do
10: \( T^k = \text{nil} \) // initialize tour \( k \) as empty
11: mark all jobs as unscheduled by ant \( k \)
12: \( \text{n} = 0; \text{feasible} = \text{true} \)
13: while \( \text{n} < \text{number.of.jobs} \land \text{feasible} \) do
14: \( \text{n} = \text{n} + 1; \text{i} = \text{job}_{n-1} \)
15: repeat
16: \( C = \text{domain}(\text{job}_n) \)
17: if \( \text{random}() > p_j \) then
18: choose next job \( j \in C \) to be scheduled by ant \( k \) with probability
19: \( p_j = \frac{\tau_{i,j}^\alpha \cdot \eta_{i,j}^\beta}{\sum_{j \in C} \tau_{i,j}^\alpha \cdot \eta_{i,j}^\beta} \)
20: else
21: end if
22: feasible = \( \text{post}(\text{job}_n = j) \)
23: if \( \neg \text{feasible} \) then
24: \( \text{post}(\text{job}_n \neq j) \)
25: end if
26: until feasible \( \land C = \emptyset \)
27: if feasible then
28: \( T^k = \text{append}(T^k, (i, j)) \)
29: \( \tau_{i,j} = (1 - \rho) \cdot \tau_{i,j} + \rho \cdot \tau_0 \) // local evaporation
30: mark job \( j \) as scheduled by ant \( k \)
31: end if
32: end while
33: if feasible\( (T^k) \) then
34: \( l_k = \text{length}(T^k) \)
35: else
36: \( l_k = \infty \)
37: end if
38: end for
39: \( \text{ibest} = \text{arg min}_k (l_k) \) // index of shortest tour
40: if \( l_{\text{best}} < l_{gb} \) then
41: \( T^{gb} = T^k; l_{gb} = l_{\text{best}} \)
42: \( \text{post}(\text{end.number.of.jobs} < l_{gb}) \)
43: end if
44: \( \forall i,j \in T^{gb} : \tau_{i,j} = (1 - \rho)\tau_{i,j} + Q \cdot \text{makespan}(T^{gb})^{-1} \) // evaporate and reward
45: end for

4.2.2 Domain Filtering

To render the search procedure more effective the given lookahead is important. Increasing the change of constructing a feasible solution increases the change of finding good quality solutions. There are various ways to increase the lookahead that is given during the search. For example, the constraint model could be augmented by redundant constraints to achieve a more effective constraint propagation and more domain pruning. A different approach is the usage of a bounding method. Bounding methods allow to exclude areas from the search space which cannot contain an optimal solution for the problem. The general idea is to calculate an upper bound on the solution while a solution is constructed. All parts of
the search space for which the lower bound exceeds the upper bound on the solution do not have to be considered by the search procedure, as they cannot contain solutions which are better than the already known one. A way to calculate this lower bounds is to solve a relaxation of the original problem. This relaxation can be seen as a simplified version that is easier to solve than the original problem, where its simplicity must be payed by getting only a lower bound on the solution of the original problem.

Having a constraint programming assisted search procedure it is relatively straightforward to utilize the information given by the relaxation. Namely, by adding an additional constraint that aims at removing all values from a variable domain which cannot improve the best solution found so far. This procedure has been termed domain filtering and was introduced in the context of solving TSPs by Focacci et al. [1999] and later applied to TSP-TW problems [Focacci et al., 2002b].

For the sake of this project we adapt the ideas presented by Focacci et al. to the Ant Colony / CP context. As a relaxation for the TSP problem the Assignment Problem (AP) is considered. The AP can be translated into the graph theoretic problem of finding a set of disjunct tours in a directed graph such that each node is visited exactly once. The relaxation is given by the fact that sub-tours are allowed, i.e., not all cities have to be connected through a unique path. Figure 4.2 exemplifies this simplification.

To solve the AP standard linear programming can be used. For this project we used the public domain linear programming library lpsolve [Berkelaar et al.] to model and solve it.

Relaxed Problem

The linear program that models the assignment problem is given by:

\[
\begin{align*}
\text{minimize} & \quad \sum_{i,j} c_{ij}x_{ij} \\
\text{subject to} & \quad \sum_{j \neq i} x_{ij} = 1 \quad \forall i \\
& \quad \sum_{i \neq j} x_{ij} = 1 \quad \forall j \\
& \quad x_{ij} \in [0,1]
\end{align*}
\]

\(x_{ij}\) are binary variables that denote if the edge between \(i\) and \(j\) is used or not. The first constraint states that all nodes \(i\) (see AP graph in Figure 4.2) must be assigned to exactly one \(j\) node, that has a different label. The second constraint models the same for the \(j\) nodes. Each \(j\) gets exactly one \(i\) node assigned, where the label of the node \(i\) must

\(^{2}\text{In out application domain this bound is simply the best solution which has been found so far during a run of the algorithm.}\)

\(^{3}\text{It is the core problem of the TSP-TW (Single Machine Job Scheduling). Although, it accounts not for the TW constraints, the AP relaxation has to be proven to be applicable in the TSP-TW context [Focacci et al., 2002a].}\)
Figure 4.2: TSP Relaxation – The graph on the left hand represents the assignment problem. Each city $i$ must be assigned a city $j$ which represents its successor. The graph on the right hand side is a more convenient representation of the same information. The edge costs are not annotated in the AP graph but are the same as in the graph on the right. Bold edges represent realized edges. Figure (a) represents the optimal solution if the problem would be considered as a real TSP, whereas Figure (b) displays the optimal solution of the AP (sub-tours are allowed).

be different from the label of the respective $j$ node. The last constraint simply states that the decision variables have to be binary. In the actual model that used in lp-Solve we relax this to a real valued constraint ($0 \leq x_{ij} \leq 1$) to be solvable with the simplex method. This is save to do because the assignment problem can be regarded as a simplified transshipment problem where all feasible solutions are integer valued (Integrality Theorem [Cormen et al., 2001]).

$c_{ij}$ denotes the cost that is associated with using edge $(i,j)$. In our problem domain this corresponds to setup time between job $i$ and job $j$.

Solving this linear program yields a lower bound for the modeled TSP problem. The objective value comprises only a lower bound on the total optimal setup times needed to schedule all jobs, it does not account for duration of individual jobs or waiting times that might be introduced by time window constraints. Nevertheless, as the solution of the AP provides a complete assignment of jobs it is easy to calculate these missing quantities. How this is done is described further below in the context of using the obtained bound to prune domain values.

Besides a lower bound and an assignment of jobs the linear programming solution of the AP also provides reduced cost information for each edge. These reduced costs correspond to the amount the objective function would increase if the belonging edge would be used.
Its calculation is based on so called dual values that are associated with each problem constraint. Solving the linear program automatically gives these values as a by-product. In our scenario we have constraint on the number of outgoing and the number of incoming edges at each node, the reduced costs for an edge \((i, j)\) can therefore be calculated by:

\[
rc_{ij} = c_{ij} - dual_{\text{outgoing}}_i - dual_{\text{incoming}}_j
\]

If an edge is part of the optimal solution of the AP than the following equation holds

\[
c_{ij} = dual_{\text{outgoing}}_i + dual_{\text{incoming}}_j
\]

Edges that are part of the solution have therefore a reduced cost of zero. In the notion of the reduced costs this is reasonable as it determines the amount the optimal solution would increase if an edge would be used, if an edge is already part of the optimal solution this edge is already used and cannot increase the objective value any further.

Having a lower bound on the solution and a quantity that gives the potential increase of this value if certain edges are used allows us to formulate a constraint that is capable of removing unpromising values from the domains of job slot variables.

\[
LB + rc_{ij} < \text{best solution found so far}
\]

If the calculated lower bound \(LB\) plus the reduced costs for an edge \((i, j)\) is bigger than the best solution that has been found so far, using this edge cannot lead to a solution that is better than the one already found. This edge has therefore not to be considered.

We add this constraint to the existing CPACS algorithm by augmenting the domain function in line 16 in Algorithm 4. The domain of the slot variable \(job_n\) is normally only determined by the underlying CP solver. Our implementation attempts to prune this domain further by excluding values that violate the above constraint. Given \(i\) from line 14 we check for all values \(j \in \text{domain}(job_n)\) if the constraint

\[
LB + rc_{ij} < l_{gb}
\]

holds. If it is violated, job \(j\) is removed from the domain of \(job_n\).

It is important to note that the underlying lp-model changes while the construction of tours proceed. Every time a job is scheduled, i.e, assigned to a slot, the corresponding edge is fixed in the lp-model. That means that additional constraints are added. For example, if the last scheduled job was \(x\) and now job \(y\) would be scheduled, then the edge \((x, y)\) would be fixed in the lp-model by adding constraints\(^4\) that state that \(x\) has an outgoing edge to \(y\) and \(y\) gets an incoming edge from \(x\).

### Consideration of Waiting Times

The lp model takes only setup times between jobs into account. The obtained solution gives therefore only a lower bound on the optimal job scheduling in terms of setup time. When this bound is used to filter additional domain values by comparing it to the best solution found so far, the duration of jobs and potential waiting times have to be added. This is because the value for the best so far found solution corresponds to the makespan of the scheduled jobs, which includes includes, apart from the setup times, the duration of the jobs and waiting times that might be introduced if a jobs finished before the release time of its successor is reached. To make the lower bound comparable to the actual solution value we add waiting times to the bound and subtract the total duration of all jobs from

\(^4\)In the actual implementation we simply set the upper bounds of \(\{x_{ij} | i = 1 \ldots n, j = 1 \ldots n, i \neq x \land j \neq y\}\) to zero.
the solution value (right hand side of the inequality). The added waiting time is calculated based on the jobs that are already fixed in the so far constructed tour. Furthermore, we consider the potential waiting time that would be introduced if a job \( j \) would be scheduled after the last scheduled job \( i \).

### 4.2.3 Dynamic Heuristic Values

The decision which job is scheduled next by an ant is in ACS based on two factors. Pheromone \((\tau)\) present at the edge to the job and its associated heuristic value \((\eta)\). The probability of using edge \((i, j)\) is given by:

\[
p_{ij} = \frac{\tau_{i,j}^\alpha \cdot \eta_{i,j}^\beta}{\sum_{j \in C} \tau_{i,j}^\alpha \cdot \eta_{i,j}^\beta}
\]

where \(C\) denotes the set of unscheduled jobs. As integral part of the reinforcement learning the pheromone values change over time to account for the learned desirability of a certain edge, whereas the heuristics are static values that reflect a particular feature of the job the edge points to, e.g., its due date or the setup time it needs to change from job \( i \) to \( j \). The heuristics would therefore, for example, prefer a job that has a smaller setup time to a one with a higher setup time. A drawback of using these heuristics is that they are static, i.e., they do not change while a tour is constructed. This might guide the search to bad solutions. If, for example, setup time is used as a heuristic it could occur that ‘close’ jobs are chosen regardless of their release date. If because of this procedure a job with a late release time is selected, unnecessary waiting time would be introduced.

To overcome this potential disadvantage of static heuristics we are proposing the usage of dynamic heuristics that are based on reduced cost information provided by the linear programming model. As introduced above, the reduced cost is calculated for each edge in the assignment graph and it gives information about the degree the optimal solution of the relaxed problem would increase if a particular edge would be used. Hence, this value can also be used to assign a selection bias to each edge in a manner as the heuristic values do. The advantage of the reduced costs (rc) is given by the fact that they are dynamic. Each time a job is scheduled the linear model accounts for that and is resolved. The reduced costs therefore reflect the desirability of edges given the currently assigned jobs.

**Calculation of Scaling Factor**

To be applicable for the replacement of the static heuristic values, the rc values must be scaled in order to keep the reinforcement learning effective. Considering a comparison of two edges (a and b) based on their selection probability

\[
\tau_a \eta_a^{\text{dyn}} \sim \tau_b \eta_b^{\text{dyn}}
\]

it has to be assured that it is dominated by the pheromone part, if a pheromone saturated edge is compared with a completely unsaturated edge\(^5\). Hence, the ratio of maximal pheromone to minimal pheromone must be greater than the ratio of maximal heuristic value to minimal heuristic value.

\[
\frac{\tau_{\text{max}}}{\tau_{\text{min}}} \geq \frac{\eta_{\text{max}}^{\text{dyn}}}{\eta_{\text{min}}^{\text{dyn}}}
\]

\(^5\)In this case the ACS algorithm has completely converged on a path. If the comparison would in this case dominated by the heuristic values, the pheromone present at an edge would be completely meaningless. The learned information about the problem that is expressed in the present pheromone would be irrelevant.
\( \tau_{\text{max}} \) and \( \tau_{\text{min}} \) are fixed and determined by the ACS algorithm. However, the size of \( \eta_{\text{max}} \) and \( \eta_{\text{min}} \) are influenced by the way the rc values are scaled. We used the following way to translate them into heuristic values

\[
\eta_{ij}^{\text{dyn}} = \frac{\varepsilon}{\varepsilon + rc_{ij}}
\]

where \( rc_{ij} \) denotes the reduced costs of using edge \((i, j)\). As the rc of using an edge is at least 0 (for edges part of the optimal solution), the maximal dynamic heuristic value is 1. The ratio condition therefore becomes:

\[
\frac{\tau_{\text{max}}}{\tau_{\text{min}}} \geq \frac{\varepsilon + rc_{ij}}{\varepsilon}
\]

As the pheromone value should dominate the comparison already before ants have completely converged on a path, a constant is added to the \( \tau_{\text{min}} \) value. If \( c = 0 \) the learning component only dominates the heuristics when the ants have completely converged on a path an edge with minimal pheromone concentration is compared to an edge of this path. In case \( c = 1 \) the learning component always dominates the heuristics unless the scaling factor \( \varepsilon \) is infinity.

\[
\frac{\tau_{\text{max}}}{\tau_{\text{min}} + c(\tau_{\text{max}} - \tau_{\text{min}})} \geq \frac{\varepsilon + rc_{ij}}{\varepsilon}
\]

The right hand side of the equation gets maximal if \( rc_{ij} \) is maximal. After rearranging the equation the scaling factor \( \varepsilon \) is given by:

\[
\varepsilon = \frac{rc_{\text{max}}}{\tau_{0} + c(\tau_{\text{max}} - \tau_{0})} - 1
\]

\( \tau_{\text{min}} \) is the minimal pheromone concentration which is \( \tau_{0} \) in our model. \( rc_{\text{max}} \) is the maximal reduced cost an edge can have, per definition this cost can be bounded by number of jobs times maximal setup time of any two jobs. \( \tau_{\text{max}} \) represents the pheromone that is present at a saturated edge. In the ACS algorithm, which uses 10 ants per iteration, an edge is saturated with pheromone if the amount of deposited pheromone is equal to the amount of evaporation at this edge.

\[
\frac{Q}{l} = \tau_{\text{max}} - ((1 - \rho)^{11} \tau_{\text{max}} + \sum_{i=1}^{10} (10 - \rho)^{i} \rho \tau_{0})
\]

The amount of pheromone \( \tau \) present at an edge at the steady state is therefore given by:

\[
\tau_{\text{max}} = \frac{\sum_{i=1}^{10} (1 - \rho)^{i} \rho \tau_{0} + \frac{Q}{l}}{1 - (1 - \rho)^{11}}
\]

\( l \) denotes the makespan of corresponding tour. We use an estimate of the average expected makespan of the given jobs to determine this quantity.

### 4.2.4 Coupling CPACS and ACSsr

To investigate if an algorithm that combines soft and hard constraint handling techniques could lead to an overall performance increase we implemented a loose coupling of ACSsr and CPACS. The way they have been coupled is rather simple. Both algorithms run in parallel independent of each other, each with its own pheromone matrix. At defined times they exchange their best so far found solutions. These exchange events have been
synchronized such that each time ACSsr has finished 10 complete iterations and CPACS has finished one iteration a solution is exchanged. The process is sketched in Figure 4.3.

Figure 4.3: Coupling of CPACSlp + ACSsr – interleaving scheme

The rational behind running ACSsr 10 times more often than CPACS is to equate the contribution of both algorithms. In previous comparisons, were CPACS and ACSsr performed competitive for particular problems [Meyer, 2005b], ACSsr used 10 times more iterations (5000 instead of 500) than CPACS. When it comes to an exchange of solutions the respective ‘foreign’ solution is handled differently in both algorithms. CPACS compares its own best found solution with the ACSsr one and updates it if the makespan of the passed ACSsr solution is better than its own best solution. In case that ACSsr has not found a feasible solution, it does not pass any solutions to CPACS.

The ACSsr side of the combined algorithm handles ‘foreign’ solutions slightly different. It stochastically ranks the obtained CPACS solution together with the other solutions found in the last iteration. The top ranking solution is then rewarded with pheromone as usual. The rank at which the received CPACS solution is inserted into the list of solutions is crucial. It effects the attention the CPACS solution receives in ACSsr. For the here introduced coupling we put it at the top rank to increase the chance that it gets rewarded. Another option would be to insert it at a random position to simulate a scenario in which ACSsr would have found the solution in the regular way. Examining the effect of different insertion positions was not a subject of this project and should be investigated in further studies.

The results of applying the coupled CPACSlp+ACSsr version to the used standard set of test problems can be found in the adjacent results section.

4.3 Results

This section presents the results of applying our different implementations to a set of standard problems. If not differently stated, we used the following parameter to run the algorithms: 10 ants per iteration (Line 9 Algorithm 4), 500 iterations, $\alpha$, $\beta$ and $Q = 1$, $\rho = 0.05$. The initial pheromone concentration $\tau_0$ is calculated by $\tau_0 = \frac{Q}{20 \cdot N \cdot \rho \cdot \hat{l}}$, where $N$ is the number of jobs and $\hat{l}$ an estimate of the average makespan of the problem. It is calculated by the sum of all processing times plus $(N-1)$ times the average setup time.
<table>
<thead>
<tr>
<th>Problem</th>
<th>W8.1</th>
<th>W8.2</th>
<th>W8.3</th>
<th>W20.1</th>
<th>W20.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>8326</td>
<td>5828</td>
<td>4245</td>
<td>8814</td>
<td>5497</td>
</tr>
<tr>
<td>Problem</td>
<td>W20.3</td>
<td>W30.1</td>
<td>W30.2</td>
<td>W30.3</td>
<td>RGB10.a</td>
</tr>
<tr>
<td>Value</td>
<td>4762</td>
<td>8882</td>
<td>5257</td>
<td>4998</td>
<td>3840</td>
</tr>
<tr>
<td>Problem</td>
<td>RGB16.a</td>
<td>RGB16.b</td>
<td>RGB21.9</td>
<td>RGB27.a.3</td>
<td>RGB27.a.15</td>
</tr>
<tr>
<td>Value</td>
<td>2596</td>
<td>2094</td>
<td>4607</td>
<td>9999</td>
<td>9999</td>
</tr>
<tr>
<td>Problem</td>
<td>RGB27.a.27</td>
<td>BR17.a.3</td>
<td>BR17.a.10</td>
<td>BR17.a.17</td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>1076</td>
<td>1557</td>
<td>1400</td>
<td>1057</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Solution values until labeling step were counted

The tables below have the following structure: *Makespan – Avg* refers to the average of the best makespan the algorithms have found for different problems over a number of independent runs. The associated standard deviation gives a measure for the variability of the best makespan found in each run. The column *Label* refers to the average number of labeling steps the respective algorithm needed in one run until it found a solution of a certain quality. The values until the labeling steps were counted are given for each problem in Table 4.1.

They represent solution qualities all algorithms should be able to reach easily. We used this way of counting to be able to get a better comparison of the algorithms. If we would have counted the steps needed to reach the optimal solution in each run, a better algorithm that tends to find better solutions on average might have been compared to a worse algorithm that is not able to find very good solutions. But the actual worse algorithm would perhaps appear to be more efficient in terms of needed labeling steps. For problem RGB27.a.3 and RGB27.a.15 we did not have enough data to give an estimate for the average solution quality an algorithm should be easily able to reach. The used 9999 therefore means that the given labeling steps for this problem refer to the number of steps needed to find a feasible solution.

To test if our different implementations show a truly different performance on the tested problems we apply a two sided t-Test. We test the average found makespan and the average needed labeling steps. The respective confidence values are given in the *p-value* tables. If NA is given as value the standard deviation of the tested random variables were 0.

### 4.3.1 CPACSlp

To investigate the effect the additional domain filtering has on the search efficiency we applied the CPACSlp algorithm to a set of test problems and compared its performance in terms of needed labeling steps and solution quality to the existing CPACS algorithm. We expected that with more search space pruning in CPACSlp either the number of labeling steps CPACSlp needs to reach the same solution quality as CPACS would be smaller, or that the overall solution quality would increases using CPACSlp. The results of applying CPACS and CPACSlp to a set of test problems (see Appendix A for a description) are shown in Table 4.2. The t-Test results of testing the average number of needed labeling steps and the average found solution qualities for being different can be found in Table 4.3.

---

6. Number of times one of the constrained sequence variables \( job_i \) is instantiated

7. Problems of computing a solution for these two problems due to occasional memory errors (Appendix B.2)

8. The results for CPACS are calculated based on our implementation. The results given by Meyer and Ernst [2004] refer to a version based on a Siiustus PROLOG implementation that utilized a stronger constraint solver. Refer to Appendix B.1 for a more elaborate description.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Makenpan Avg</th>
<th>Makenpan Sd</th>
<th>Label Avg</th>
<th>Label Sd</th>
<th>Makenpan Avg</th>
<th>Makenpan Sd</th>
<th>Label Avg</th>
<th>Label Sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8.1</td>
<td>8324.7</td>
<td>2.2</td>
<td>1188.83</td>
<td>1371.06</td>
<td>8324.2</td>
<td>2.45</td>
<td>1229.48</td>
<td>1564.98</td>
</tr>
<tr>
<td>W8.2</td>
<td>5829.2</td>
<td>17.11</td>
<td>10557.3</td>
<td>11292.52</td>
<td>5829.2</td>
<td>17.28</td>
<td>8262.98</td>
<td>8676.62</td>
</tr>
<tr>
<td>W8.3</td>
<td>4245.0</td>
<td>0.0</td>
<td>1655.73</td>
<td>1575.98</td>
<td>4245.0</td>
<td>0.0</td>
<td>1807.64</td>
<td>2370.82</td>
</tr>
<tr>
<td>W20.1</td>
<td>8611.98</td>
<td>56.01</td>
<td>6992.06</td>
<td>5360.55</td>
<td>8608.4</td>
<td>36.85</td>
<td>6774.1</td>
<td>5474.24</td>
</tr>
<tr>
<td>W20.2</td>
<td>5143.74</td>
<td>27.82</td>
<td>2396.43</td>
<td>1058.59</td>
<td>5141.9</td>
<td>21.74</td>
<td>2384.5</td>
<td>1336.14</td>
</tr>
<tr>
<td>W20.3</td>
<td>4439.56</td>
<td>39.42</td>
<td>4943.27</td>
<td>4020.63</td>
<td>4459.6</td>
<td>43.04</td>
<td>4383.18</td>
<td>3742.29</td>
</tr>
<tr>
<td>W30.1</td>
<td>8375.33</td>
<td>82.7</td>
<td>7233.79</td>
<td>4763.08</td>
<td>8371.5</td>
<td>78.14</td>
<td>8619.1</td>
<td>4253.19</td>
</tr>
<tr>
<td>W30.2</td>
<td>4925.4</td>
<td>94.63</td>
<td>20593.2</td>
<td>18180.32</td>
<td>4904.9</td>
<td>76.87</td>
<td>18751.0</td>
<td>11714.84</td>
</tr>
<tr>
<td>W30.3</td>
<td>4426.72</td>
<td>76.45</td>
<td>5091.49</td>
<td>3217.59</td>
<td>4424.6</td>
<td>66.27</td>
<td>4740.72</td>
<td>2742.29</td>
</tr>
</tbody>
</table>

| RBG10.a | 3840         | 0.0         | 112.512   | 9.17     | 3840         | 0.0         | 112.16    | 9.43     |
| RBG16.a | 2596         | 0.0         | 202.31    | 127.14   | 2596         | 0.0         | 233.78    | 325.63   |
| RBG16.b | 2094         | 0.0         | 4366.1    | 5365.1   | 2094         | 0.0         | 356.92    | 5086.67  |
| RBG21.9 | 4514.05      | 14.85       | 3294.8    | 1552.71  | 4512.94      | 13.93       | 3150.62   | 1679.57  |
| RBG27.a.3 | 1706.1    | 19.81       | 275.8     | 2.32     | 1698.12      | 8.09        | 275.24    | 2.34     |
| RBG27.a.15 | 1438.7   | 7.21        | 378.3     | 10.38    | 1435.8       | 10.98       | 377.32    | 13.35    |
| RBG27.a.27 | 1076     | 0.0         | 209.891   | 55.45    | 1076         | 0.0         | 182.00    | 56.17    |
| BR17.a.3 | 1526.59     | 6.01        | 3477.46   | 2484.04  | 1525.72      | 9.93        | 4184.3    | 2916.27  |
| BR17.a.10 | 1397.98    | 6.09        | 16643.3   | 14769.23 | 1398.36      | 0.77        | 19768.1   | 18844.32 |
| BR17.a.17 | 1057       | 0.0         | 183.956   | 27.61    | 1057         | 0.0         | 193.72    | 28.26    |

Table 4.2: Results of CPACS – learning, no bounding and CPACS – learning, bounding (50 independent runs)

**Discussion**

The results of the t-Test do not show a significant difference in terms of obtained solution quality or number of needed labeling steps using the CPACSlp version. For all except two problems the p-values report no difference in the tested means. For W20.3 and BR17.a.17 a true difference is indicated. CPACS appears to find on average better solutions for W20.3 than CPACSlp and to need less labeling steps to find solutions for BR17.a.17.

These observations do not meet our previous expectations. It appears as if the additional domain filtering we used has no effect or even a slightly detrimental effect.

There are different possible explanations for this rather bad performance of the CPACSlp version. The used bounds might be too weak to actually allow to prune important domain values and therefore to reduce the search space significantly. A second reasonable explanation is that the effect of the additional domain filtering is disguised by the pheromone learning effect. The influence of the learning might overwhelm the additional pruning in a way that values which are pruned in the LP version would only be selected with a small chance because others are far more desirable due to a higher pheromone concentration. A third possible explanation is that the additional pruning might exclude ‘bridging’ solutions that lead to better solutions sounds reasonable. If edges are excluded by the domain filtering, they cannot be used anymore and do not experience a local pheromone evaporation. This biases the selection probability of this edge in a negative way, if we consider that a particular edge (transition from one job to another) can appear more than once in the search tree. If this really has a negative influence on the search performance remains to be investigated in further studies. First investigations should test if the observed detrimental effect for problems W20.3 and BR17.a.17 is only a random appearance or if it persists when more independent runs are performed.

To investigate the potential disguising effect of the learning component we implemented variants of CPACS and CPACSlp that did not have a learning component and compared their performance.
4.3.2 CPACSlp without learning component

To investigate the effect the additional domain filtering would have on the search efficiency without any side effects we compared CPACS and CPACSlp with deactivated pheromone learning component. Both were applied to the same set of test problems and their performance in terms of average makespan and average labeling steps where compared. The performance on the test problems can be found in Table 4.4. The belonging t-Test results are displayed below in Table 4.5.

<table>
<thead>
<tr>
<th>Problem</th>
<th>CPACS</th>
<th>CPACSlp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MakeSpan</td>
<td>Label</td>
</tr>
<tr>
<td>Problem</td>
<td>Avg</td>
<td>Sd</td>
</tr>
<tr>
<td>W8.1</td>
<td>8323.8</td>
<td>2.48</td>
</tr>
<tr>
<td>W8.2</td>
<td>5830.5</td>
<td>13.79</td>
</tr>
<tr>
<td>W8.3</td>
<td>4245.0</td>
<td>0.00</td>
</tr>
<tr>
<td>W20.1</td>
<td>8788.9</td>
<td>34.66</td>
</tr>
<tr>
<td>W20.2</td>
<td>5475.2</td>
<td>34.08</td>
</tr>
<tr>
<td>W20.3</td>
<td>4722.2</td>
<td>29.26</td>
</tr>
<tr>
<td>W30.1</td>
<td>8808.2</td>
<td>73.33</td>
</tr>
<tr>
<td>W30.2</td>
<td>5234.6</td>
<td>42.5</td>
</tr>
<tr>
<td>W30.3</td>
<td>4953.9</td>
<td>29.39</td>
</tr>
<tr>
<td>RBG10.a</td>
<td>3840.0</td>
<td>0.0</td>
</tr>
<tr>
<td>RBG16.a</td>
<td>2596.0</td>
<td>0.0</td>
</tr>
<tr>
<td>RBG16.b</td>
<td>2094.0</td>
<td>0.0</td>
</tr>
<tr>
<td>RBG21.9</td>
<td>4585.6</td>
<td>9.39</td>
</tr>
<tr>
<td>RBG27.a.3</td>
<td>1847.47</td>
<td>8.14</td>
</tr>
<tr>
<td>RBG27.a.15</td>
<td>1485.6</td>
<td>4.32</td>
</tr>
<tr>
<td>RBG27.a.27</td>
<td>1076.0</td>
<td>0.0</td>
</tr>
<tr>
<td>BR17.a.3</td>
<td>1550.74</td>
<td>6.26</td>
</tr>
<tr>
<td>BR17.a.10</td>
<td>1398.78</td>
<td>3.33</td>
</tr>
<tr>
<td>BR17.a.17</td>
<td>1057.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 4.4: Results of CPACS – no learning, no bounding and CPACSlp – no learning, bounding (50 independent runs)

Discussion

With deactivated learning component we observe that for relatively hard problems, e.g., W20.1, the general performance of both algorithms decreases. The average solution quality decreases while the number of labeling steps needed to find these solutions increases. For easier problems, e.g., W8.1 switching off the learning component appears to make no
difference, which indicates that the algorithms are capable of exploring enough promising parts of the search space for these problems without relying on the reinforcement learning. However, both, the performance of the CPACS as of the CPACSlp algorithm seem to degrade in a similar quantity for the hard problems such that there is no significant difference between CPACS and CPACSlp in terms of obtained solutions quality and needed labeling steps. A t-Test on makespan and needed labeling steps confirms this speculation. The resulting p-values reflect no significant difference between both algorithms, neither in the average makespan nor in the number of labeling steps needed.

Having these results we can conclude that the learning component does not hide a potential additional domain pruning effect. As we cannot observe a significant difference between both algorithms it emerges that the used bounds are apparently to loose that no values, or no significant proportion of values, can be excluded from the domain of unassigned job slots.

To achieve a significant effect the calculated lower bound would have to be tighter, which means closer to the actual optimal solution of the problem. There are a number of ways how this could be achieved, refer to Section 5.1.2 in the Future Research part for a more elaborate description.

### 4.3.3 Effect of Dynamic Guidance

Given the unsatisfactory results for the additional domain filtering, we attempt to assess the effect of a dynamic guidance scheme, based on reduced cost information, could have on the search efficiency. We implemented an algorithm that used these reduced cost information to replace the static heuristic values. We ran analysis with different choices ($c \in \{0.2, 0.4, 0.6, 0.8\}$) of the scaling factor $c$ (Section 4.2.3 – Calculation of Scaling Factor). After applying the algorithm with different values for $c$ to the set of standard problems it seemed that the choice of $c$ seemed not to have a great influence on the performance. The results for $c = 0.2$ are displayed in Table 4.6. Results of testing makespan and labeling steps for difference are given in Table 4.7.

### Discussion

Our conjecture was that utilizing dynamic heuristic would improve the search performance because the reduced cost information represent a dynamic desirability of edges which depends on the previously constructed job assignment rather than a static information that is given by the problem itself.

Testing the actual performance difference between using dynamic heuristics and static heuristics implemented in two different versions of CPACSlp show that they appear not
to make a significant difference. Furthermore, it has been observed (data not shown) that for different values of \( c \) the generated solution quality does not change. This insensitivity to the setting of the parameter could be caused by two reasons. The calculated scaling factor \( \varepsilon \) (Section 4.2.3) could be wrong because the estimate for the maximal reduced cost coefficient might overestimate the actual reduced costs that are normally observed for the given problems. Assuming overestimation, the pheromone learning component would overwhelm the dynamic heuristics such that no significant effect can be observed. Another explanation could be the choice of the parameter. It is possible that an effect is only observable if the adjustment parameter \( c \) is smaller than 0.2, as we started with 0.2 we are likely to overlook something.

The further investigation of these two possibilities appears to be rather straightforward and should definitely attempt in future studies.

Table 4.7: t-Test results (p-Values)
### 4.3.4 Coupling CPACS and ACSsr

Table 4.8 represents the results for the loose coupling algorithm. Column *best* refers minimal makespan that was found for the respective problem in 20 independent runs of the algorithm. *Average Makespan* and *Sd* give the average over the found makespan and its standard deviation.

<table>
<thead>
<tr>
<th>Problem</th>
<th>CP(ACS)$^2$sr</th>
<th>CPACS</th>
<th>ACSsr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Sd</td>
<td>Avg</td>
</tr>
<tr>
<td>W8.1</td>
<td>8322.25</td>
<td>2.17</td>
<td>8324.7</td>
</tr>
<tr>
<td>W8.2</td>
<td>5818</td>
<td>0.0</td>
<td>5829.2</td>
</tr>
<tr>
<td>W8.3</td>
<td>4245</td>
<td>0.0</td>
<td>4245</td>
</tr>
<tr>
<td>W20.1</td>
<td>8522.25</td>
<td>23.2</td>
<td>8611.98</td>
</tr>
<tr>
<td>W20.2</td>
<td>5103</td>
<td>16.09</td>
<td>5143.74</td>
</tr>
<tr>
<td>W20.3</td>
<td>4327</td>
<td>15.98</td>
<td>4439.56</td>
</tr>
<tr>
<td>W30.1</td>
<td>8147.26</td>
<td>155.48</td>
<td>8375.33</td>
</tr>
<tr>
<td>W30.2</td>
<td>4633</td>
<td>40.88</td>
<td>4925.4</td>
</tr>
<tr>
<td>W30.3</td>
<td>4262.25</td>
<td>33.44</td>
<td>4262.72</td>
</tr>
<tr>
<td>RBG10.a</td>
<td>3840</td>
<td>0.0</td>
<td>3840</td>
</tr>
<tr>
<td>RBG16.a</td>
<td>2596</td>
<td>0.0</td>
<td>2596</td>
</tr>
<tr>
<td>RBG16.b</td>
<td>2094</td>
<td>0.0</td>
<td>2094</td>
</tr>
<tr>
<td>RBG21.9</td>
<td>4499.85</td>
<td>8.74</td>
<td>4514.05</td>
</tr>
<tr>
<td>RBG27.a.3</td>
<td>1682.35</td>
<td>19.26</td>
<td>1706.1</td>
</tr>
<tr>
<td>RBG27.a.15</td>
<td>1433.15</td>
<td>10.71</td>
<td>1438.7</td>
</tr>
<tr>
<td>RBG27.a.27</td>
<td>1076</td>
<td>0.0</td>
<td>1076</td>
</tr>
<tr>
<td>BR17.a.3</td>
<td>1514.65</td>
<td>21.69</td>
<td>1526.59</td>
</tr>
<tr>
<td>BR17.a.10</td>
<td>1396.6</td>
<td>6.1</td>
<td>1397.98</td>
</tr>
<tr>
<td>BR17.a.17</td>
<td>1057</td>
<td>0.0</td>
<td>1057</td>
</tr>
</tbody>
</table>

**Table 4.8:** CPACS + ACSsr – Makespan results (pf = 0.7, 20 independent runs for CP(ACS)$^2$sr), Results for CPACS and ACSsr taken from above, 50000 tour constructions in ACSsr, 10 independent runs for ACSsr

**Discussion**

To assess the potential of combining hard and soft constraint handling techniques in one algorithm we attempt a loose coupling of CPACS and ACSsr. Applying the algorithms to our standard problem set revealed that the CPACS + ACSsr algorithm in respect to each single algorithm did not show a significant performance improvement. It is capable of generating the respective best result its components achieve for given problems, but it does not improve the overall performance. This outcome does not allow to infer that a coupling of soft and hard constraint handling techniques is not worthwhile. The gathered results are inconclusive. For a number of problems the coupled versions appears to have a small benefit for others the opposite can be observed. To examine these effects further a statistical analysis with more independent runs should be performed to get more reliable data.

Considering a stronger constraint solver the benefit of using CPACS solutions in ACSsr might even increase (Appendix B.1 explains the weakness of the utilized solver). The given implementations do not show a complementary strength for different problems (except RBG27.a.27) as they did in [Meyer, 2005b]. Coupling algorithms that exhibit such a complementary strength for certain problems still remains an important area of study.
Chapter 5

Conclusions

In this thesis we investigated different ways of handling constraints in Ant Colony Optimization. We could show that the usage of Stochastic Ranking (SR) as a soft handling technique in ACO is worthwhile. A replacement with a simpler method which was intended to have a similar effect did not achieve as good results as SR.

Furthermore, we investigated possible improvements of the CPACS algorithm using bounds on the solution which have been obtained by solving a relaxation of the actual TSP-TW problem. It emerged that the calculated bounds were not tight enough to have an significant effect on the efficiency of the search. The potential additional domain filtering gained by using reduced cost information appears not to prune the search space in a way that the search process itself becomes more efficient. Investigating the tightness of the calculated bound and a possible strengthening are therefore immediate steps which would have to be attempted. Due to the fact that we were not able to get the results for the algorithms at an earlier stage (Appendix B.2), we could not achieve this improvements by ourselves, yet.

Moreover, this thesis explored a possible replacement of static heuristic values by reduced cost information gathered from solving a relaxation of the actual TSP-TW problem. We explored that with the applied scaling scheme the dynamic heuristic values make no difference regarding the efficiency of the tested algorithms. However, it is likely that the search efficiency will improve once a more appropriate scaling factor is found and the quality of the utilized bound is refined.

Finally, we attempt a loose coupling of hard and soft constraint handling techniques in ACO. We could not observe an overall performance improvement for the coupled version, rather an effect that the coupled version generated the respective best solution its compartments found for a certain problem. A conjecture in this respect is that with a stronger constraint solver (as used in the original CPACS version) the strength of CPACS in solving tight problems would emerge more strongly in contrast to ACSsr and therefore it would become more likely that a coupling would lead to a benefit.

Overall we observed that using computational less expensive techniques such as Stochastic Ranking to handle constraints is the better choice if the given problems are only weakly or moderately constrained. The utilization of CP techniques seems only to pay off for highly constrained problems where SR struggles to find feasible solutions. To get a better impression of the applicability of different algorithms to different problems the actual executing times should be compared rather than the number of needed labeling steps. See next section for a related discussion.
5.1 Future Research

5.1.1 Labeling Steps – Computation Time Tradeoff

In the present study we compared the algorithms based on labeling steps needed to find a solution. This gives a measure for the amount of search that is carried out as it reflects the number of variable labeling steps until the best solution of a run is found. It does not give information about the computational efficiency of the algorithms. In fact, CP assisted algorithms might need less labeling steps to find a good solution than simpler algorithms, but they are computational far more expensive. Investigating this computational overhead is an important area of study, considering potential practical applications of these algorithms. If a simpler algorithm is capable of generating the same solution quality by carrying out more search in less time than a more complex, search efficient, one, it will be preferred. To get a fair comparison between different concepts highly efficient implementation of them would be needed to avoid a biasing of the results by different quality implementations.

5.1.2 Tightness of Bounds

We have observed that the bounds obtained solving the relaxation of the problem appear not to be tight enough to have significant influence on the pruning of domain values. The first step to investigate is which values are actually pruned and how tight the bounds are. Meaning, the calculation of the gap between calculated bound and actual solution value in each step of a tour construction. This information will help to develop strategies to improve the used relaxation model.

The current implementation adds duration times and eventual idle times to the bound after it has been calculated. Incorporating these into the actual relaxation model might improve the quality of the bounds as the optimal solution of the relaxation would be aware of them. Another possible improvement incorporates the augmentation of the linear programming model by using additional constraints that, for example, prohibit direct cycles in the relaxed solution \((x_{ij} + x_{ji} \leq 1)\). With this extension obtained lower bound would become more accurate. There a vast number of other possible improvements that could be made regarding the used relaxation model. A detailed investigation of those in connection with our problem setting would require a number of further studies.

5.1.3 Efficient Relaxation Solving

If it should turn out that tightening the solution bounds leads to a significant improvement of the search efficiency one might consider to use the primal-dual Hungarian Algorithm [Carpaneto et al., 1988] to solve the relaxation of the problem (Assignment Problem). It is capable of solving the Assignment Problem in \(O(n^3)\), and can solve subsequent recomputations in \(O(n^2)\).

5.1.4 Insertion of Exchanged Solutions

In the loosely coupled algorithm CPACS and ACSsr exchange solutions at defined time points. The rank SR inserts the solution it got from CPACS were chosen to be the first in the current implementation. It might be worthwhile to investigate the effect of inserting the foreign solution at a random rank in the list of solutions. This would not overrate the CPACS solution. It would be equivalent to a solution that SR just encounters during its normal search.

Another possible variation would be to exchange infeasible solutions. ACSsr could also give its best infeasible solution to CPACS. Increasing the likelihood for CPACS to generate
infeasible solutions appears to be counterproductive as CPACS can only generate feasible solutions, but rewarding edges that belong to an infeasible solution might also increase the change to find good feasible solutions.

5.1.5 More Learning

The described algorithms conduct 5000 independent searches. The information one search learns about the problem is given by changing the pheromone concentration of certain edges. Edges that were part of a good solution become more likely to be reused. The change of using bad edges decreases. What that type of learning does not accomplish is the fact that bad edges, meaning edges that certainly lead to an infeasible solution, are ultimately excluded. In fact, the constraint system should actually exclude this type of edges as it is aware of possible constraint violations but as we have seen, with a weak constraint solver this is not always the case. Using information that is gathered using the search process might therefore proof to be useful. For example, if we would know that a certain combination of variable assignments will always lead to an infeasible solution we could exclude this particular assignment from future use, e.g, in the remaining $n$ independent searches. This could be achieved by posting additional constraints to the model that describe the prohibited assignments.
Appendix A

Test data

We used single machine job scheduling problems which were drawn from two different sources. Problems starting with the letter ‘w’ (first 9 problems in the common tables) originate from an Australian wine bottling plant. The setup times are actual change over times of the wine bottling robot. These jobs do not have a release time, as it is considered that the wine is always available in tanks. Due dates of the jobs are randomly selected from a set of possible dates during a planning period. The processing times or durations of the jobs are artificially generated to create problems of varying tightness. The duration of each job is randomly chosen from the interval \([\frac{1}{2}\mu, \frac{3}{2}\mu]\), where \(\mu\) represents the average processing time (calculated by maximal due date - number of jobs * average step time) and \(\beta\) is a constant that is either 1.2, 2 or 3 depended on the assigned problem class 1, 2 or 3. Problems from class 1 are therefore the tightest ones, whereas problem class 3 represents the loosest problems (a set sampled from \([\frac{1}{6}\mu, \frac{3}{2}\mu]\) will contain more problems that get close to the latest due date than a set sampled from \([\frac{1}{6}\mu, \frac{1}{2}\mu]\)). The first number in the problem name indicates the number of jobs, the second number the problem class. For example w8.1 refers to the tightest problem with 8 jobs.

The other problems were taken from the ATSP-TW literature. They involve for example scheduling of a stacker crane in a warehouse. Not all jobs in these data sets have associated time windows. The number of jobs in the set is given by the first number were the second number represents the number of jobs that have associated time window constraints. For example RBG 21.9 contains 21 jobs, were 9 have an associated time window.
Appendix B

Obstacles

B.1 Problem with all_different Constraint / Weak Solver

Some problems were encountered while we reimplemented the existing Sicstus PROLOG version of CPACS in Java using ILOG Jsolver 2.0. The decision of reimplementing CPACS in Java were made beforehand for several reasons. We considered it to be easier to modify and add possible extensions to the algorithm. To implement the constraint modeling part, we chose to use the commercial constraint programming library ILOG JSolver. Regarding the usual good reputation of ILOG, we expected this library to show a good performance in terms of propagation strength equal to Sicstus Prolog. Unfortunately, it turned out that this was not the case. We discovered that the propagation strength of the provided constraint solver is rather weak. This results in a lack of provided lookahead and we were therefore not able to regenerate solutions of the same quality as reported in [Meyer and Ernst, 2004].

The most outstanding shortage were the implementation of the all_different constraint\(^1\). We discovered that the constraint does not fail when it actually is expected to fail. That is in situations were the number of unassigned variables is bigger than the number of values that is left to assign, a rather trivial case were the solver should immediately fail. The following example illustrates the dilemma.

\[
D := \{1, 2, 3, 4, 5\}
\]

\[
x_1 \in D \\
x_2 \in D \\
x_3 \in D \\
x_4 \in D \\
x_5 \in D
\]

\[
t \in \{1, 2, 3, 4, 5\} \\
t \neq x_1 \\
t \neq x_2 \\
t \neq x_3
\]

\(^1\)This constraint is usually posted on a number of constraint variables and states that they all have to take different values.
\[ t \neq x_4 \]
\[ t \neq x_5 \]
\[ t = 3 \]

After posting all these constraints the domain \( D \) of the \( x \) variables narrows down to:
\[ D = \{1, 2, 4, 5\} \]

That means there are only 4 values left for 5 variables that share the same domain. The constraint solver should immediately recognize it and fail. Instead JSolver starts to assign values to the \( x \) variables until it discovers at some point that no values are left.

The reference manual of JSolver claims that it would behave in the expected way (fail immediately), for that reason this substantial drawback was unfortunately discovered rather late in the developing process. A request at ILOG support resulted in a hint that the documentation is ambiguous and will be changed in later versions.

Investigating the reasons for the bad performance of our Java implementation cost us a couple of weeks which would have been spent on a more efficient implementation of the algorithm, as it was outlined in the Research Proposal for this project. An even bigger problem was that we did not have a strong solver that we expected to have in the beginning.

### B.2 Severe Cluster Instability

This project was concerned with the analysis and development of algorithms with a stochastic nature. To assess the performance of different implementations and extensions of such algorithms they had to be run multiple times on the same test instances to get a reliable statistical measurement. In order to carry out such computational rather expensive experiments in a reasonable amount of time it is inevitable to have access to high performance computing facilities. When the project started we assumed that these facilities were available in the Faculty of IT. Unfortunately, we had to discover that these facilities broke down. In the last two month of the project we experienced severe problems with the cluster and its Nimrod interface. Disappearing experiments, database errors, nodes suddenly missing the Java VM are only a couple of examples. The described problems made it impossible to obtain the desired results before the actual due date of this thesis.

Fortunately, we were granted the permission to carry out our computations at a high performance cluster at the Max Planck Institute for Molecular Genetics in Berlin-Dahlem. In that respect, special thanks to Dr. Rainer Spang and the people at the Computational Molecular Biology department.

Again is hard to give an estimate how many hours were wasted by restarting experiments, correspondence with administrators and adjusting scripts to run at the recovery cluster. But it is evident that this time would have been spend wisely on the project.
References


Bernd Meyer. Personal communication, 2005a.


