AGH University of Science
and Technology
in Krakow

M.Sc. Thesis

Constraint Programming for Stochastic Problems

Anna Michalak

Studies: Mathematics
Specialization: Financial Mathematics

Faculty of Applied Mathematics

2007

Acknowledgements

I would like to express the sincerest thanks to my supervisor Professor Chris Beck. Thank you for give me an opportunity to work in your laboratory, it was great experience for me. Thank you for your encouragement, support and guidance.

I would also like to thank Daria Terekhov, for always being very helpful during my stay at the University of Toronto. Thank you for many useful discussions, for explaining me lots of things, support, big help with improving my English and for your friendship.

Contents

Abstract

The main aims of this dissertation are to present current constraint programming techniques for stochastic problems, give an overview of the optimization problems in queueing theory and examine a possibility of applying constraint programming techniques to these problems. We present an alternative constraint programming model for a problem based on queueing theory, compare its performance to the model presented in previous work as well as examine an extension of another queueing-based problem presented in literature.

Key words

constraint programming, stochastic programming, queueing theory, optimization under uncertainty

Introduction

In most mathematical programming problems it is assumed that parameters occurring in the model are constant and known: such models are called *deterministic*. However, deterministic models are usually just an approximation of the reality. These models can be used to solve the problem in one situation (for one specific set of data values), but the solutions obtained from these models may become sub-optimal or infeasible if the situation changes. Sometimes variability of the parameters is so significant and their evaluation so uncertain (especially if the parameters will be assigned values in the future) that treating the deterministic model as a good approximation is not acceptable. Uncertainty in decision problems comes from, for example, weather changes, market-related uncertainty and competition.

To deal with such situations, a variety of techniques has been developed. Some of these techniques, such as stochastic constraint programming (chapter 2) and constraint programming combined with queueing theory (chapters 4 and 5) will be discussed in this dissertation.

This dissertation consists of five chapters:

1. Constraint Programming
   In this chapter we describe constraint programming and the problems it is used for. In particular, we define the Constraint Satisfaction Problem (CSP), present properties of constraints and different approaches for solving $CSPs$, and discuss methods for finding the optimal solution.
2. Stochastic Constraint Programming
   This chapter introduces a technique of solving problems which involve uncertainties: Stochastic Constraint Programming (SCP). We present two alternative semantics developed so far to solve problems modelled by means of SCP.
3. Queueing Theory
   In the Queueing Theory chapter, we present the main definitions related to this field. We also present an overview of models for the optimal design and control of queues.
4. Solving a Queueing Design and Control Problem with Constraint Programming. $LogPSums$ model
   A problem first described in a paper by Berman et al. [37] is presented. Previously

proposed constraint programming models created for this problem are described, and a new model, the *LogPSums* model, is presented. Experimental results are analyzed and the new model is compared to the best model from previous work.

5. Switching time
   An extension of the problem described in the previous chapter is presented. A method for solving this problem is proposed, and experimental results are shown in order to demonstrate the correctness and usefulness of this method.

Chapter 1

Constraint Programming

1.1. Constraint programming

Constraint Programming (CP) is the study of computational systems based on constraints. The main idea of constraint programming is to solve problems by stating constraints (conditions, properties or requirements) about the problem area and, subsequently, finding a solution satisfying all of the constraints [5]. The earliest ideas leading to CP may be found in Artificial Intelligence (AI), dating back to sixties and seventies. Logic Programming (LP) has been also noted to be just a particular kind of CP [5]. CP techniques are used to solve hard combinatorial problems and are very competitive with techniques from Operations Research (OR). CP is now a mature field and has been successfully used for tackling a wide range of complex problems from real-life applications.

In order to solve a given problem by means of CP, we need to formulate it as a Constraint Satisfaction Problem (CSP). This phase of CP is called modelling.

Definition 1.1.1. A Constraint Satisfaction Problem (CSP) is described as a triple $X, D, \mathcal{C}$ where:
- X is a finite sequence of variables $(x_1, x_2, \ldots, x_n)$,
- D is a set of domains $D_1, \ldots, D_n$, where D_i is a finite set of values that can be assigned to variable x_i; values in the domains can be integers, reals, Booleans or symbols. An assignment is a pair (x_i, a_i), such that the variable $x_i \in X$ is assigned the value $a_i \in D_i$,
- a finite set of constraints, $\mathcal{C} = (c_1, \ldots, c_e)$, where $X(c_j) \in X$ is the set of variables involved in constraint c_j [4].

Each constraint c_i involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assign-

ment of values to some or all of the variables, $\{x_i = v_i, x_j = v_j, ...\}$. An assignment that does not violate any constraints is called a consistent assignment. A solution to a CSP is an assignment in which every variable is mentioned and all constraints are satisfied.

The CSP can be represented as an undirected graph, called the *constraint graph*, where nodes represent variables, and edges represent constraints.

1.1.1. Constraints

A constraint is simply a logical relation among several unknown quantities (or variables), each taking a value in a given domain. The constraint thus restricts the possible values that the variables can take. Constraints have several interesting properties [5]:

- constraints may represent partial information, i.e. the constraint need not uniquely specify the values of its variables. For example, the constraint $X > 2$ does not specify the exact value that the variable X can take: X can be equal to 3, 4, 5, etc.

- constraints are heterogeneous, i.e. they can bind unknowns from different domains, for example the length (number) with the word (string) [6].

- constraints are non-directional, i.e. two variables X, Y can be used to infer a constraint on X given a constraint on Y, and vice versa. For example, $X = Y + 2$ can be used to compute the variable X using $X \leftarrow Y + 2$ as well as the variable Y using $Y \leftarrow X - 2$.

- constraints are declarative, i.e. they specify what relationship must hold without specifying a computational procedure to enforce that relationship; the constraints become part of a program which describes and solves the problem.

- constraints are additive, i.e. the order of imposition of constraints does not matter, all that matters in the end is that the conjunction of constraints is in effect [5].

- constraints are rarely independent, i.e. typically constraints used in the model share variables. Constraints may be described in the form of database relations, propositional logic clauses, equations, inequalities.

- constraints may be global. A global constraint is a single constraint on any number of variables which is used to replace a set of constraints. Global constraints have been developed to replace particular sets of constraints that occur frequently.

1.1.2. Representation

There is often more than one way to model a problem. To choose among the several natural representations (decision variables, constraints) of a given problem, proper understanding of the underlying alternatives and occasionally some good insights are

required. This aspect of modelling is very important since the way the problem is modelled has a direct effect on how quickly the problem can be solved. In other words, different problem representations lead to search trees of different size, which is a good measure of the complexity of these representations. However, a smaller number of variables and simpler constraints do not necessarily imply that the problem can be solved more quickly. It is important to also consider using global constraints. These constraints allow a single constraint on any number of variables to replace a set of constraints and have an efficient domain reduction algorithm [7]. Global constraints are one of the key factors in the success of constraint programming. They help users model problems as they specify patterns that frequently occur in the real world. Efficient constraint propagation algorithms associated with global constraints help solve the problem.

Examples of problem for which constraint programming techniques have been successfully applied are: the n-queens problem, crossword puzzles, map coloring, k-colorability, scheduling problems [2].

1.2. Different approaches developed for solving $CSPs$

Basic approaches to solving $CSPs$ are:

- search algorithms (backtracking, systematic search),
- inferential consistency algorithms (constraint propagation),
- combination of these two techniques.

1.2.1. Systematic search

Generate and Test is the most trivial form of systematic search and the most basic way to solving $CSPs$. Generate and Test is a paradigm where each possible combination of the variables is systematically generated and then tested to see if it satisfies all the constraints. The first combination that satisfies all the constraints is a solution [5].

More efficient methods use a backtracking paradigm. Backtracking is an algorithm that explores the search space and backtracks to the previous state when a "dead end" is encountered. If it still does not lead to a solution then the algorithm backtracks further, etc. One of the major drawbacks of the standard backtracking scheme is thrashing because of lack of node/arc-consistency (see section 1.2.2) [5].

This drawback can be avoided by intelligent backtracking, that is, by a scheme in which backtracking is done directly to the variable that caused the failure. Another major drawback is performing redundant work due to fact that this algorithm does not learn from the failure of different nodes.

1.2.2. Consistency techniques and constraint propagation

Although sophisticated search algorithms play an important role in solving constraint-based problems, constraint propagation (domain filtering, pruning or consistency techniques) is the main advantage of CP.

Classical consistency algorithms were first explored for constraint networks in artificial intelligence research in the late 1960s. Consistency-based algorithms use information from the constraints to reduce the search space as early in the search as it is possible.

The main algorithms:

- *Node consistency* is the simplest of consistency techniques. It removes values from variables' domains that are inconsistent with unary constraints on respective variables [5].

- *Arc consistency* is the most widely used consistency technique; it removes values from variables' domains that are inconsistent with binary constraints [5].

  Definition 1.2.1. The arc (V_i, V_j) is *arc consistent* if and only if for every value x in the current domain of V_i which satisfies the constraints on V_i there is some value y in the domain of V_j such that $V_i = x$ and $V_j = y$ is permitted by the binary constraint between V_i and V_j.

  Arc consistency algorithms - algorithms that are used to reduce variable domains by excluding inconsistent values from the variable domains. Arc consistency ensures that any legal value in the domain of a single variable has a legal match in the domain of any other selected variable.

- *Path consistency algorithms* - algorithms that are used to reduce variable domains by excluding inconsistent values from the variable domains. Path consistency ensures that any consistent solution to a two-variable subnetwork is extensible to any third variable. More generally, algorithms that ensure that any consistent solution to subnetworks having i variables are called $i - consistency\ algorithms$ [2].

- $i - consistency\ algorithms$ - these algorithms guarantee that any consistent instantiation of $i - 1$ variables is extensible to any ith variable. If a network is $i - consistent$ for all i, we call it *globally consistent*.

V. Kumar in his survey [8] presents and considers various algorithms for solving $CSPs$. Constraint propagation can be used to solve $CSPs$ without search but as the author notices: "Although, any n-variable CSP can always be solved by achieving $n - consistency$, this approach is usually even more expensive than simple backtracking". On the other hand, $CSPs$ can always be solved by the standard backtracking algorithm, although at substantial cost. A third scheme is to embed a constraint prop-

agation algorithm inside a backtracking algorithm. By performing constraint propagation, a given CSP is essentially transformed into a different CSP whose search space is smaller. In the work of Kumar [8], a combination of different algorithms which reduce the search space using backtracking is studied carefully. The optimal combination of these techniques is different for different problems and is still a topic of investigation.

Constraint Satisfaction Problems are known to be NP-complete, and, therefore, general-purpose polynomial algorithms are clearly unavailable. Efficiency can be improved by bridging the gap between CP and the other communities such as Operations Research, Local Search, Planning and Machine Learning. The thrust of all constraint processing is to develop algorithms that work well in a wide range of problem classes.

1.3. Constrained optimization problems ($COPs$)

Branch and bound is a natural method to solve constrained problems but is not efficient and not sufficient to solve large problems. Local search methods have also been used to solve constrained optimization problems ($COPs$). A constrained optimization problem is defined as a CSP together with an objective function. In general, the methods of searching for the global minimum/maximum of an objective function can be divided into two groups [11]:

- Heuristic methods which find the global minimum only with certain probability. The most common are:

  – Stochastic Algorithms and Genetic Algorithms, e.g. Tabu Search - a method used to avoid cycling and getting trapped in a local minimum. It is based on the notion of a tabu list, that is, a special short-term memory that maintains a selective history, composed of previously-encountered configurations. If this configuration is on the tabu list, then it is not allowed to be used for some number of iterations. Such a strategy allows the search process to go beyond a local optimum.

  – Simulated Annealing - a related global optimization technique which traverses the search space by generating neighboring solutions of the current solution. A superior neighbor is always accepted. An inferior neighbor is accepted probabilistically based on the difference in quality and a temperature parameter. The temperature parameter is modified as the algorithm progresses in order to alter the nature of the search.

  – Evolutionary Algorithms (EAs) - search methods that take their inspiration from natural selection and survival of the fittest in the biological world.

- Methods which let us find the global minimum with certain precision. They are based on the $branch - and - bound$ technique - splitting the feasible region into smaller feasible subregions and finding upper and lower bounds for the optimal

solution within a feasible subregion.

There are other techniques used for solving Constraint Satisfaction Optimization Problems, such as dominance rules and shaving.

1.3.1. Dominance rules

Dominance rules are constraints that forbid compound assignments that would lead to a solution that is dominated by another one. This means that we are sure that there must be another solution that is equally good or better [4]. Dominance rules have been defined by Ibaraki [14] as binary relation defined on the set of partial problems generated in a $branch - and - bound$ algorithm.

These constraints are not logical consequences of the constraints C and do not necessarily preserve the set of optimal solutions [4]. These constraints add efficiency to the $branch - and - bound$ algorithm. They have been proven to be useful in solving scheduling problems.

1.3.2. Shaving

Shaving is a constraint programming technique. In order to make a problem easier to solve, some redundant constraints may be added temporarily. Combining these added constraints with the original ones, a stronger deduction can be achieved. A reduction ("shaving") of the domains of the variables is performed by applying the whole set of constraints. This process helps to remove inconsistent values from the domains at the beginning of search, which in many cases can reduce the search tree by a lot. Shaving can also be incorporated into the $branch - and - bound$ algorithm and reduce the domains after each decision [40].

1.4. Applications

Constraint Programming techniques are widely applied in such fields as: computer graphics (expressing geometric coherence in the case of scene analysis, drawing programs), natural language processing, database systems, operations research problems (time-tabling, planning and scheduling, resource allocation), molecular biology (DNA sequencing, chemical hypothesis reasoning), business applications (option trading), electrical engineering (to locate faults), circuit design (to compute layouts) [5].

Chapter 2

Stochastic Constraint Programming

2.1. Introduction to Stochastic Constraint Programming

Most models of mathematical programming problems assume constant parameters. Such models are called *deterministic* and are just an approximation of the reality which is always affected by uncertainty. Deterministic models can solve the problem in only one situation (for one specific set of data values). However, if the situation changes, then the deterministic model may not be correct anymore. In order to deal with problems affected by uncertainty, Stochastic Constraint Programming (SCP) was created. Stochastic models, in general, deal with reality better than deterministic models, which are not always even a good approximation of the real situation.

SCP extends constraint programming to deal with both decision variables, which can be set by the decision maker, and stochastic variables, which follow some probability distribution function. The area of research in the topic of SCP is very new - the earliest publication about SCP [7] was in 2002. Stochastic constraint programming is inspired by both stochastic integer programming and stochastic satisfiability. The most significant advantages in this approach are derived from the best features of traditional constraint satisfaction, stochastic integer programming, and stochastic satisfiability. For example, we are able to write more complex, and at the same time more expressive, models using non-linear constraints, global and arithmetic constraints and to exploit efficient constraint propagation algorithms.

To solve Stochastic Constraint Programs, two alternative semantics have been developed so far: policy-based [7] and scenario-based [16], [19]. These methods are still being improved and compared by researchers.

Since many combinatorial problems that arise in real-life applications involve uncertainty, the development of effective constraint programming methods for dealing with uncertainty is very important.

2.1.1. Stochastic Constraint Satisfaction Problem

In similar way as we defined a Constraint Satisfaction Problem, let's define a Stochastic Constraint Satisfaction Problem.

Definition 2.1.1. A *Stochastic Constraint Satisfaction Problem* ($SCSP$) is a 6-tuple: < V, S, D, P, C, θ > where [19]:

- V is a set of decision variables,
- S is a set of stochastic variables,
- D is a function mapping each element of V and each element of S to a domain of potential values,
- P is a function mapping each element of S to a probability distribution for its associated domain,
- C is a set of constraints, where a constraint $c \in C$ on variables $x_1, \ldots, x_j$ specifies a subset of the Cartesian product $D(x_1) \times D(x_2) \times \ldots \times D(x_j)$ indicating mutually-compatible variable assignments. The constraints of C that constrain at least one variable in S are called chance constraints.
- θ_h is a threshold probability in the interval $[0, 1]$, indicating the fraction of scenarios in which the chance constraint h must be satisfied. In an m-stage stochastic CSP, sets V and S are partitioned into m disjoint sets: $V_1, \ldots, V_m$ and $S_1, \ldots, S_m$. Each ith stage consists of sets: V_i and S_i, $i = 1, \ldots, m$.

Each constraint is defined as in a traditional constraint satisfaction problem as a logical relation among several variables, each taking a value in a given domain. Constraints can be over decision variables or over decision and stochastic variables. Both hard constraints and chance constraints occur in $SCSPs$.

A hard constraint is a constraint that must be satisfied with threshold 1, i.e. in all the possible scenarios that may occur. A *scenario* (also called *a world* or *one of the possible worlds*) is a set of values for the stochastic variables.

A chance constraint c_i needs to be satisfied in at least a fraction, θ_i, of the world.

A $feasible\ solution$ is an assignment of a value from its domain to every variable which satisfies all of the constraints.

2.2. Two different approaches to solve $SCPs$

So far, two alternative approaches to solving $SCSPs$ have been proposed: *policy- based procedures* that extend standard methods from CP and *scenario- based methods* that solve $SCSPs$ by reducing them to a sequence of $CSPs$.

Depending on the method used, assumptions may be different. In order to apply the policy-based approaches, one has to assume that only one global chance constraint is present, and that all stochastic variables occurring in the problems are independent. On the other hand, the authors of the articles about the scenario-based approach have

relaxed the assumption about independence of the stochastic variables, and added multiple chance constraints as well as a range of objective functions like maximizing the down-side.

2.3. Policy-based approach to solve $SCPs$

One of the approaches developed by Walsh [7] is based on the exploration of the policy space. A *policy* is a tree (tree of decisions) with nodes labelled with value assignments to variables, starting with the values of the first variable in X labelling the children of the root, and ending with the value of the last variable in X labelling the leaves. Policies determine how decision variables are set depending on earlier decision and stochastic variables.

Leaf nodes take value 1 if the assignment of values to variables along the path to the root satisfies all the constraints and 0 otherwise. Each path to a leaf node in a policy represents a different possible *scenario* (set of values for the stochastic variables) and a different set of values given to the decision variables in this scenario. The number of scenarios in the $SCSP$ grows exponentially with the number of stages in the problem.

The probability of each world is defined as:

$$\prod_i P(s_i = d_i)$$

where:

- s_i is the ith stochastic variable on a path to the root,
- d_i is the value given to s_i on this path,
- $P(s_i = d_i)$ is the probability that $s_i = d_i$.

The satisfaction of a policy is a sum of the leaf values weighted by their probabilities.

- We say that a policy satisfies the constraints if and only if its satisfaction is at least θ. In this case we say that a policy is satisfying.

- A stochastic constraint satisfaction problem is satisfiable if and only if there is a policy which satisfies the constraints (a satisfying policy).

- The optimal satisfaction of a stochastic CSP is the maximum satisfaction of all policies.

- For a stochastic constrained optimization problem, the expected value of a policy is the sum of the objective valuations of each leaf node weighted by their probabilities.

- If a policy satisfies constraints and maximizes (or minimizes) the expected value, then this policy is the optimal policy.

In a *one-stage stochastic CSP* the decision variables are set before the stochastic variables. This problem is satisfiable if and only if there exist values for the decision

variables so that, given random values for the stochastic variables, the probability that all the constraints are satisfied equals or exceeds a threshold θ.

In a *two-stage stochastic CSP*, there are two sets of decision variables: V_{d_1}, V_{d_2} and two sets of stochastic variables: V_{s_1}, V_{s_2}. The aim is to find values for the variables in V_{d_1}, so that given random values for V_{s_1}, we can find values for V_{d_2}, so that given random values for V_{s_2}, the probability that all the constraints are satisfied equals or exceeds θ. An important remark here is that the value chosen for the second set of decision variables V_{d_2} is conditioned on both the values chosen for the first set of decision variables V_{d_1} and on the random values given to the first set of stochastic variables V_{s_1} [16]. An m-stage $SCSP$ is defined in analogous way.

The most common ways of solving problems stated in such a way are *backtracking* (BT) and forward checking (FC). These algorithms are extensions of the backtracking and forward-checking algorithms that are used in deterministic CP. By means of these algorithms we are able to find the optimal policy.

2.3.1. Backtracking

BT finds an assignment to the decision variables which gives an optimal satisfaction θ. BT is called with the *search depth*. This procedure first puts the decision variables which occur together in order (if they are not already). This is done by means of the "fail first" heuristic, which chooses first the variables that are most likely to fail. At the beginning we set the bounds for θ: θ_h and θ_l. These upper and lower satisfaction bounds are used to prune search. If the optimal satisfaction lies between these bounds, BT returns the exact satisfaction. If the current assignment to the decision variables returns a satisfaction of $\theta > \theta_l$, then any other values must exceed θ to be part of a better policy. Reduction of the search space is made if the current θ is greater than θ_l; in this case, we replace the lower satisfaction bound by $max(\theta, \theta_l)$. Policies with lower satisfaction are simply rejected [7].

2.3.2. Forward checking

The forward checking procedure is an extension of the backtracking algorithm. Forward checking in general is a technique for evaluating the effect of a specific assignment to a variable. Given the current partial solution and a candidate assignment to evaluate, it checks whether another variable can take a consistent value. In other words, it first extends the current partial solution with the tentative value for the considered variable; it then considers every other variable x_k that is still unassigned, and checks whether there exists an evaluation of x_k that is consistent with the extended partial solution. More generally, forward checking determines the values for x_k that are consistent with the extended assignment [20].

Results of experiments performed by Walsh [7] show that the FC algorithm clearly dominates the BT algorithm. One of the main reasons for this is that the FC algorithm checks forward and prunes values from the domains of future decision and stochastic variables which violate the constraints. *Consistency checking* and *domain pruning* ensure that the algorithm only visits a small fraction of the possible worlds.

Forward checking fails if:

- a stochastic or a decision variable has a domain "wipe-out",
- a stochastic variable has so many values removed that we cannot hope to satisfy the constraints.

We denote q_i as an upper bound on the probability that the values left in the domain of stochastic variable s_i can contribute to a solution. If FC ever reduces q_i to be less than θ_l, it backtracks, as it is impossible to set s_i and satisfy the constraints adequately [15]. When forward checking removes some values d_j from s_i then the reduction on q_i is made. FC reduces q_i by the value of the probability $P(s_i = d_j)$. However, in the cases when backtracking occurs, a special procedure is called. This procedure restores values which have been removed from future variables and resets the value of q_i for the stochastic variables.

2.3.3. Improved FC algorithm

The forward checking algorithm seems to work well, but it has some drawbacks. Balafoutis and Stergiou [18] developed the FC algorithm by identification and correction of a flaw in the procedures proposed by Walsh [15].

Balafoutis and Stergiou [18] prove that when the current variable is a stochastic variable, there are cases when the algorithm should continue going forward instead of backtracking even if values of q_i have been reduced to be less than θ_l. The reason why such states can occur is that the satisfaction of future sub-problems may contribute to the total satisfaction, and the current policy has a chance to return a satisfaction that is greater or equal to the satisfaction lower bound.

However, if the current variable is a decision variable and q_i (for some future stochastic variable) is reduced by FC to a value $q_i < \theta_l$, there is no possibility for the total satisfaction to be greater than the lower satisfaction bound.

In Figure 2.1, we can see a graphic explanation of the described problem: we see that the original algorithm proposed by Walsh backtracks (because $q_i < \theta_l$) and returns the maximum satisfaction incorrectly.

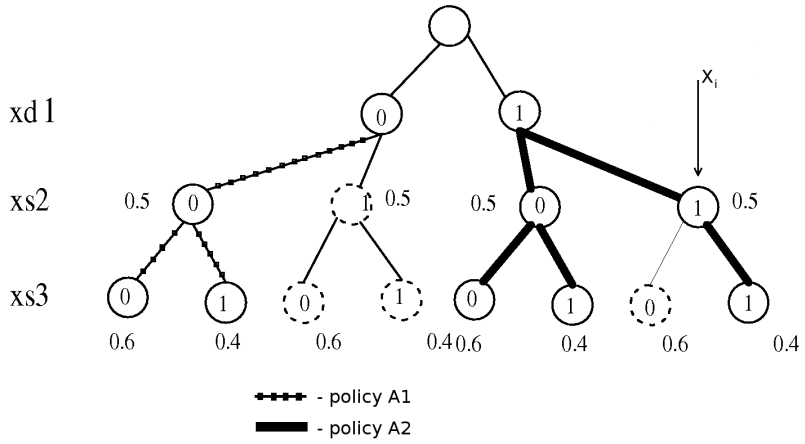

Figure 2.1. Search tree.

Define x_{d1} to be a decision variable, and x_{s2}, x_{s3} to be stochastic variables. Suppose that all three variables have a domain $\{0, 1\}$.

When the current stochastic variable is x_i (as marked in Figure 2.1), the algorithm reduces q_i by the value of probability $P(x_{s3} = 0)$ because there is a constraint disallowing the tuple $< x_{s2}, x_{s3} >=< 1, 0 >$. As we can see now, $q_i = 0.4$, and 0.4 is less than the lower bound of 0.5.

The algorithm now backtracks and chooses policy A1, whose satisfaction is equal to 0.5. This choice is incorrect because the policy A2 has a greater satisfaction, equal to 0.7 (because: $0.5 \cdot 0.6 + 0.5 \cdot 0.4 + 0.5 \cdot 0.4 = 0.7$).

The main point in correcting the flaw and improving FC is that even if $q_i < \theta_l$, we cannot be sure that the satisfaction of the currently explored policy cannot be greater than the threshold. Therefore, we need to check whether the maximum satisfaction offered by the current stochastic variable is enough to lift the total satisfaction over θ_l or not. The current assignment can be safely rejected only if the sum of the quantities of:
- the already computed satisfaction of the previously-assigned values of the current variable,
- the maximum satisfaction of the sub-tree below the current assignment and
- the sum of the probabilities of the following values of the current variable (i.e. the maximum satisfaction that they can contribute) is lower than θ_l. Otherwise, continuation of the expanding should be done.

Balafoutis and Stergiou [18] incorporated the function "check" into the FC algorithm. This procedure checks and eventually removes values from the domains of future variables. In this way, the search space is reduced.

Example 2.3.1. Consider the problem of finding the maximum satisfaction with one decision variable, x_{d1}, and two stochastic variables, x_{s1} and x_{s2}, each with domain $\{0, 1\}$ [18].
Define:
$P(x_{s2} = 0) = 0.2$, $P(x_{s2} = 1) = 0.8$, $P(x_{s3} = 0) = 0.8$, $P(x_{s3} = 1) = 0.2$.
Suppose that the constraints of the problem disallow the following tuples: $< x_{d1}, x_{s2} >=< 0, 0 >$, $< x_{d1}, x_{s2} >=< 1, 0 >$, $< x_{d1}, x_{s3} >=< 0, 1 >$, $< x_{d1}, x_{s3} >=< 1, 1 >$.

Assume that the total satisfaction computed for $x_{d1} = 0$ is equal to 0.64. The algorithm checks the total satisfaction for the assignment $x_{d1} = 1$. After taking the constraints into account, we compute the total satisfaction for this policy, and it happens to be equal to 0.64 as well. The algorithm visited 6 nodes - these are marked in grey in Figure 2.2. However, we can reduce the number of visited nodes to 4 by comparing the total satisfaction of the previous policy with the current one. While being in node $x_{d1} = 1$, the FC algorithm determines the total satisfaction of the current assignment by computing the probabilities of further stochastic variables:

$$\prod_{s=i+1}^{n} \sum_{t=1}^{|D(x_s)|} P(x_s = v_t)$$

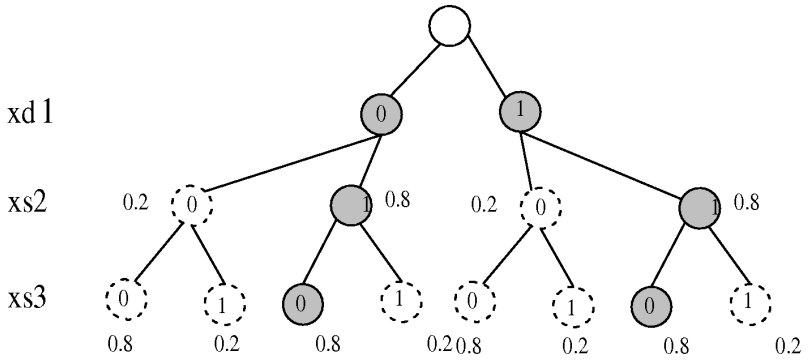

Figure 2.2. Search tree.

weighted by probability $P(x_i = v_j)$ if x_i is stochastic.

In our example, $x_i = x_{d1}$ (which is a decision variable); $v_t = 1$; $|D(x_s) = \{0, 1\}| = 2$; $n = 2$.

If the total satisfaction of the current assignment happened to be smaller than the previous one, then the algorithm would not have to visit further nodes. This happens in our example:

$$\begin{aligned}\prod_{s=i+1}^{n} \sum_{t=1}^{|D(x_s)|} P(x_s = v_t) &= \sum_{t=1}^{2} P(x_{s2} = v_t) \sum_{t=1}^{2} P(x_{s3} = v_t) \\ &= P(x_{s2} = 1)P(x_{s3} = 0) = 0.8 \cdot 0.8 = 0.64.\end{aligned}$$

The total satisfaction of the current assignment $x_{d1} = 1$ is equal to the previous one, 0.64 (for $x_{d1} = 0$). Therefore, the algorithm does not have to visit further nodes.

2.3.4. Approximation procedures

Besides complete algorithms (BT, FC) for solving $SCSPs$, several techniques for approximating the answer were proposed by Walsh [15]:

- replacing stochastic variables by their most probable values (or by their median in ordered domains) leads to a traditional constraint satisfaction problem (or a constraint optimization problem if we are dealing with a stochastic COP),
- Monte Carlo sampling of stochastic variables,
- local search over policy space ($GSAT$, $WalkSAT$).

2.3.5. Arc consistency (AC) algorithm for $SCSPs$

Balafoutis and Stergiou [18] introduce an arc consistency algorithm for solving stochastic $CSPs$ (see section 1.2.2).

Definition 2.3.1. A value $v \in D(x_{d_i})$ (where $D(x_{d_i})$ is domain of decision variable x_{d_i}) is consistent if and only if there is a satisfying policy, in which at least one scenario includes the assignment v for the variable x_{d_i}.

The AC algorithm used during search may identify some of the values of decision variables for which the current policy cannot achieve satisfaction that equals at least the current lower bound θ_l. This means that these values are arc inconsistent and can be safely pruned without worrying about pruning any feasible solution.

2.4. Scenario-based approach

This semantics is based on a decision tree in which each path represents a different possible scenario and the values assigned to decision variables in this particular scenario. A scenario is a set of values for the stochastic variables. The most important advantage of applying this approach to solve $SCSPs$ is the possibility of using existing, very efficient constraint solvers or hybrid solvers. This is because using the scenario-based approach we can compile a stochastic constraint problem down into a non-stochastic, "regular" constraint problem. We can treat each scenario as a separate constraint problem.

This approach may be hard to apply in problems with many stochastic variables and many stages. The reason is that the number of scenarios grows exponentially with the number of stages. The cost of computation of all possible scenarios depends, of course, on their quantity. To avoid high cost, in many practical problems a reduction of scenarios is often performed. There are lots of techniques to reduce the number of scenarios. These are useful for solving many real-world problems. The most common examples of such techniques are:

- replacing stochastic variables by their expected values and then considering just one scenario,
- taking into account just the most probable scenarios and ignoring the rest,
- Monte Carlo sampling,
- Latin hypercube sampling,
- a method which heuristically deletes scenarios to approximate as closely as possible the original scenarios according to a Fortet-Mourier metric on the stochastic parameter space [19].

2.4.1. Robust solutions

The idea of robust solutions insists on identical decisions in different scenarios. Robust solutions are solutions which satisfy the constraints regardless of what values the stochastic variables take. In some problems, this kind of solution can be desirable. In others, it is enough to ensure that only a subset of the decision variables is assigned

robust values. An important fact is that the cost of finding the robust solutions is high, and therefore the profitability of considering robust solutions should be taken into account.

2.5. Comparison (policy-based vs. scenario-based)

General remarks about the policy-based approach:

- the algorithms proposed by Walsh [15], BT and FC, can be used for binary problems only,
- changing the number of chance constraints (from a single global constraint to multiple ones) requires making large changes in backtracking and forward checking algorithms.

General remarks about the scenario-based algorithm:

- the scenario-based approach is not limited to binary problems,
- the most significant advantage of this approach is that it can exploit existing advanced CSP solvers (with specialized propagation techniques),
- the number of scenarios in $SCSPs$ grows exponentially with the number of stages in the problem. Therefore, it may not be possible to apply this approach to problems with many stochastic variables and many stages,
- a scenario-based approach permits conditional probabilities, multiple chance constraints and a wide range of goals.

As we can see, more advantages come from using the scenario-based approach to solving SCP problems than from using the policy-based approach.

Chapter 3

Queueing Theory

3.1. Introduction to Queueing Theory

Queueing theory is the mathematical study of waiting lines (or queues). A queueing system can be met anywhere. It is easy to recall a situation when waiting was required, for example, in shops, in banks, in a library. There are also some less obvious examples of queues, such as waiting for the phone connection with the extension number, waiting for the change of traffic lights or for the morning mail. All of these situations have a common feature, namely, the arrival process of people or objects that need to be served. If people or mechanisms which serve customers are occupied when a customer arrives, then the customer has to wait for his/her turn. Such customers form a queue.

In order to describe a particular queueing system, one has to create an accurate model. Models are usually divided into two types: *descriptive*, which describe the current real-world situation (for example, how long one has to wait in the queue), and *prescriptive*, which prescribe what should be done in order to achieve a particular situation (for example, what should be done to minimize customer waiting time). Most research which has been done in the area of queueing theory concerns descriptive models. However, there has also been some work done in the field of optimization problems which prescribe "the optimal course of action to follow"[1]. This area of research is referred to as the *optimal design and control of queues*. In general, design and control models deal with determining the optimal system parameters (what they should be) e.g. the optimal service rate or the optimal number of servers (channels). Which parameters are to be optimized depends on how the system has been modelled and which parameters are under control.

3.2. Queueing System Characteristics

Any queueing system can be described in terms of the following parameters [1]:

- Service pattern.
  In order to describe the sequence of customer service times, a probability distribution is needed. The service process may depend on several factors. For example, it may depend on the number of customers waiting for service since servers may work faster/slower when the queue is building up.

- Number of servers (s).
  We can have single or multiple server systems. Multiple server systems can have a single queue (e.g. a hair-styling salon where customers are waiting for the next available hair-stylist) or a queue for each server (e.g. a supermarket where customers are waiting in line to one particular cash register).

- Queue discipline.
  Queue discipline describes the order of customer service. $First\ Come$, $First\ Served$ ($FCFS$) is the most frequently-used discipline. Other disciplines include: $Last\ Come$, $First\ Served$ ($LCFS$); selection for service in random order independent of the time of arrival to the queue (RSS); a variety of *priority* schemes (for example, customers with a higher priority are served first regardless of their arrival time).

- Arrivals.
  The process of arrivals is stochastic. We need to know the probability distribution describing the times between successive customer arrivals [1]. There are several aspects that may be different for different queueing models. For example, there are many patterns of customer arrivals. We can take into account the situation when some customers waiting in the queue lose their patience and decide to leave, or, in contrast, we can consider a queue in which all customers wait until they are served. Another thing which differs among the queues is that customers can arrive singly or in groups (batch or bulk arrival). Population of customers can be finite or infinite.
  The simplest arrival process is one where we have completely regular arrivals (i.e. the same constant time interval between successive arrivals). Frequently, it is assumed that if arrivals are random, then they follow a Poisson process. In a Poisson stream, successive customers arrive after intervals which are independent and exponentially distributed. As mentioned in [42], "the Poisson stream is important as it is a convenient mathematical model of many real life queuing systems and is described by a single parameter - the average arrival rate".

3.2.1. Kendall's Notation for Classification of Queue Types

To classify a queueing system into different types, standard notation proposed by D. G. Kendall is used [35]. According to this notation, systems are described as follows:

$$A/B/C/D/E$$

where:

A	Distribution of inter-arrival times of customers
B	Distribution of service times
C	Number of servers
D	Maximum total number of customers which can be accommodated by the system
E	Calling population size

A and B can be of any of the following distribution types:

M	Exponential Distribution (Markovian)
D	Degenerate (or Deterministic) Distribution
Ek	Erlang Distribution (k = shape parameter)
G	General Distribution (arbitrary distribution)

In descriptive queueing models, parameters C, D and E can be 1, any integer greater than 1 or infinity. In queueing design and control models, these parameters can be variables. For cases where D or E are assumed to be infinite, they are usually omitted in the notation.

Some examples of queueing systems are:

- $D/M/n$ - system has n servers, service times follow an exponential distribution and the arrival process is assumed to follow a degenerate distribution.
- $Ek/El/1$ - this would describe a queue with an Erlang distribution for the inter-arrival times of customers (with a shape parameter of k), an exponential distribution for service times of customers (with a shape parameter of l), and a single server.
- $M/M/s/K/N$ - this system has s servers, each server has an independently and identically distributed exponential service time distribution, the arrival process is assumed to be Poisson and the system can accept only K customers. In this model, a maximum of K customers are allowed to stay in the queueing system at once, and there are N potential customers in the calling population.

3.2.2. Little's formula

In this section we present some notation and a very important relationship in queueing theory - Little's Formula.

We need to define several parameters that we will use further:
λ - mean arrival rate (number of arrivals per unit of time),
μ - mean service rate per server,
s - number of servers,
L - average number of units (customers) in the system (includes both the customers waiting for service and those being served),
L_q - average number of customers in the queue (waiting to be served),

W - average waiting time in the system,
W_q - average waiting time in the queue (service time not included).

In queueing problems, we often desire to find the probability distribution for the total number of customers in the system at time t, $N(t)$.

Let $p_n(t) = P(N(t) = n)$ be the probability of having n customers in the system at time t, and $p_n = P(N = n)$ be the steady state probability of having n customers in the system. The system is said to have reached a *steady state* when it has been running long enough so as to settle down into some kind of an equilibrium position [42].

The average number of customers in the system is $L = E[N]$. For a system with s servers, the steady-state average number of customers in the system is equal to

$$L = \sum_{n=0}^{\infty} np_n,$$

and the steady-state average number of customers in the queue is:

$$L_q = \sum_{n=s+1}^{\infty} (n-s)p_n.$$

One of the most powerful relationships in queueing theory has been developed by John D. C. Little (1960). He related the steady-state mean system size to the steady-state average customer waiting time.

Let T denote the total time a customer spends in the system. Then $T = T_q + S$, where T_q represents the time a customer spends waiting in the queue before being served, and S is the service time. The mean waiting time in the queue is $W_q = E[T_q]$, and the mean waiting time in the system is $W = E[T]$. These are the most frequently used measures of system performance with respect to the customers.

Little's formula:

$$L = \lambda W$$

and

$$L_q = \lambda W_q$$

A proof of this formula can be found in the work of Little [36].

We can explain this formula intuitively: if the average customer spends T time units in the system, and the arrival rate is λ, then there are, on average, λT customers waiting in the system [1].

These formulas, as well as the formula $W = W_q + \frac{1}{\mu}$, are useful when values for some of these parameters need to be calculated. Using the above formulas, we can describe the average number of customers in service when the system is in a steady state using the following relation [1]:

$$L - L_q = \lambda(W - W_q) = \lambda(\frac{1}{\mu}) = \frac{\lambda}{\mu}.$$

3.3. Optimal Design and Control of Queues

In design problems, once the value of a controllable parameter has been determined, it stays fixed and one can easily calculate this system's characteristics, such as the expected customer waiting time or the expected number of customers in the system.

Control models are dynamic and the goal in such problems is to determine when and how to change the arrival or service rates in order to optimize some objective function [27]. This is usually equivalent to determining the length of the queue at which the server should start or stop serving. A rule which prescribes the action to be taken when the queueing system is in a particular state is called a *policy*.

We illustrate the differences between queueing design and queueing control problems using the following example [1]:

Example 3.3.1. $M/M/s/K$ problem - the system has s servers; each server has an independently and identically distributed exponential service time distribution; the arrival process is assumed to be Poisson and the system can accept only K customers. In this problem, the optimal truncation K $(1 \leq K \leq \infty)$ needs to be found in order to maximize the expected profit rate. In this problem, the single value K has been assumed to be the "optimal" policy. Finding the value of K is considered a design problem. On the other hand, it is possible that a different policy can be better. For example, a policy which says that the system starts turning customers away after the K_1th customer joins the queue and a new customer can join the queue only if queue size drops to K_2, $(K_1 > K_2)$ may be better. Determining which of the two policies is better is a queueing control problem.

In the two following sections, major work done in the area of queue design and control is presented. Problems with different controllable parameters are considered.

3.3.1. Optimal Design of Queues

Several economic models and techniques for queueing design optimization problems have been studied by F. Hillier in the paper "Economic models for industrial waiting line problems"(1963) [21]. We can divide these problems into three general classes, where the following parameters need to be determined:
1. the number of servers, s,
2. the number of servers, s, and the arrival rate, λ,
3. the number of servers, s, and the service rate, μ.

Restrictions for the parameters are as follows:
- $0 < \lambda < s\mu$,
- $C_w > 0$,
- $C_s > 0$,

where C_w is the cost of waiting per unit time, and C_s is the marginal cost of service per unit of time. The first restriction says that customers who join the queue must be served at least as fast as they arrive [24], which ensures that the system can reach the steady state. If this restriction is not satisfied, the average number of arrivals into the system will exceed the maximum average service rate and the queue will continue

getting bigger with time. Such a queue never settles down and can never be in a steady state.

The problem considered by Brigham in his work "On a Congestion Problem in an Aircraft Factory" (1955) [22] is a good example of a problem of type 1. The goal of the problem is to determine the number of clerks that need to be placed behind service counters in a Boeing Airplane factory. This problem is solved by Brigham under the general condition that the number of service counters is equal to 1 using a $trial-and-error$ method. However, with more difficult problems, this method does not work very well.

As an illustration of the second type of queueing design problems, consider an industrial building with a big population of employees [21]. This population requires some service, such as rest rooms, medical centers, canteens, tool cribs, etc. It is necessary to determine how many such facilities (and how many servers) should be distributed throughout the population. In order to answer this question we need to find out what proportion of the population, i.e. what value of λ, should be assigned to each service facility. In order to simplify the problem, we can assume that λ and s (the number of servers) are the same for all service facilities.

It turns out that 1 is the optimal number of service facilities. However, this solution has been proven to be ridiculous and unreasonable in some cases: for example, one rest room in the entire Pentagon. The reason for failure of this model is the negligence of the cost of time wasted in travelling to and from the facility. Therefore, another model has been developed which takes into account the cost of the time wasted in travelling to and from the facility. In order to estimate this expected travel time, several $Travel\ Time\ Models$ which differ in the way they compute the mode of travel were presented. The most common model in industrial settings is one which describes travel as movement along a system of orthogonal aisles.

In the third type of queueing design problems, a balance between the mean cost of waiting and the service cost needs to be found. Morse [24] provides us with a motivating example of this type of problem by considering a harbour where ships are unloading.

Another example of an optimal design problem appears in the work of Evans [25], which considers the problem of optimizing the steady-state probabilities of a Markov chain with respect to an unknown parameter μ. As the author states, the problem of designing queueing systems can be formulated as one of choosing the optimal one from among a number of alternative finite state aperiodic Markov chains having a single class of ergodic states according to the criterion of expected average cost. In general this optimization problem can be written as follows:

$$optimize \quad c(\mu) + (f, P(\mu)), \quad \mu \in M$$

$$s.t. \quad P(\mu)(Q + \mu W) = P(\mu)$$

where:
- μ is a design parameter,
- M is some set,
- $c(\mu)$ is a cost associated with the control (with the parameter μ),

- f is a vector of costs or profits,
- $Q + \mu W$ is the Markov operator if μ is the design parameter, Q, W - matrices of constraints,
- $P(\mu)$ is the vector of steady state probabilities (assumed to exist for all μ),
- $(f, P(\mu))$ is a cost dependent on the steady-state probabilities.

The main idea in the approach taken by Evans is the derivation of a formula for the derivatives of the objective function with respect to service rate μ. The formula for the derivative of stable probabilities of Markov chains is derived and a gradient algorithm is suggested by Evans to find locally optimal chains. In general, the representation of the derivatives with respect to the design parameter helps to create more algorithms to solve design problems.

3.3.2. Optimal Control of Queues

As mentioned previously, the goal of optimal queue control problems is to determine the policy which minimizes cost or maximizes profit.

One of the approaches to optimize the cost is to determine a *stationary policy*. A *stationary policy* is a type of policy which prescribes the action that should be taken when the system is in a particular state, independent of time. The action may be to turn the server on and off in order to minimize such costs as the server operating cost or the cost related to customer waiting time.

In most queueing models, the number of servers is assumed to be constant. However, in many real life situations, the number of servers varies, depending on the number of customers waiting in the queue. Moder and Phillips [26] consider the optimal control of queues model in which the number of operating servers is being increased by one each time the length of the queue reaches a particular threshold M_1. If the size of the queue falls below the critical value M_2, then the added servers are removed, one at a time. There is a limited number of servers that can be added.

In the work of Moder and Phillips, there is no explicit formula for the cost function. As Gross [1] noticed, there is no consideration related to any other policy which could be better, more effective. Therefore, this model can be classified as a descriptive one. Effectiveness is compared by checking the various values of the parameters (c_1, M_1, c_2, M_2). Effectiveness can be determined by many different measures. Among the most common are measures such as the fraction of time that the servers are idle, the mean waiting time in the queue, and the mean length of the queue.

The threshold (or control limit) policies

One type of stationary policy is a *threshold policy*. This kind of policy specifies the length of the queue for which the server is switched on, under the assumption that the server can be turned off only if the queue is empty.

The three most frequently discussed types of threshold policies are the N-policy, the D-policy and the T-policy. An overview of the most important results on this topic is provided by L. Tadj and G. Choudhury [27].

N-policy

One of the most common approaches in the field of queue control is to apply N-policy. Its main idea is to turn the server on if the number of customers in the queue grows to N and to turn it off when the system becomes empty. This policy deals with the single-server systems. One of the examples of applying this policy is provided by Yadin and Naor [29]. They studied an $M/G/1$ queue where the service station is dismantled if there are 0 customers waiting in the queue and re-established when there are R customers waiting in the queue. It can be seen that this is the N-policy, with $R = N$. In terms of optimization problems, the contribution of the various components to the total cost is assumed to be linear with respect to their average values. Each of these components depends on R.

There are many different questions related to the N-policy. Details on this topic can be found in the review of Tadj and Choudhury [27].

In particular, the applicability of the N-policy can be studied for queueing systems where

- customers can arrive singly or in groups,
- we can have single or bulk service,
- there can be many different types of input or service time distributions,
- there can be vacationing/non-vacationing servers,
- there can be finite or infinite system capacity,
- priority or non-priority queues,
- impatient or non-impatient customers.

D-policy

Another type of policy, called the D-policy, consists of turning the non-operating server on when the workload reaches or exceeds a level D and turning it off when the system becomes empty. The notion of "workload" refers to the sum of service times of all waiting customers, which is the total time that the server would need to spend serving the customers who are already waiting in the queue [31]. One of the examples of using this policy is presented by Ho Woo Lee and Jung Woo Baek in their paper "BMAP/G/1 queue under D-policy: queue length analysis" (2005) [32]. The authors studied the queue length distribution of a queueing system with $BMAP$ (the batch Markovian arrival process) arrivals under the D-policy.

Quite interesting conclusions related to the superiority of different policies are derived by Tadj and Choudhury [27]. J. R. Artalejo in his work "A Note on the Optimality of the N- and D-policies for the $M/G/1$ Queue" (2002) [33] proved the fact that the superiority of the policies depends on system parameters and either one of the two policies (N or D) may be superior.

T-policy

In the above-mentioned policies monitoring the system by the server is made continuously. If this is impossible then a T-policy can be used. Under this policy, the system is scanned T time units after the last busy period in order to determine if there are any customers in the system. If some customers have arrived, then the server is turned on to start serving them (the busy period starts); otherwise, we say that a busy period

of length zero starts. In both cases, the next scan is made T time units after the end of a busy period.

In general, when trying to solve the optimization problem using one of the mentioned policies, one has to find the threshold value, such as N, D, or T which minimizes the objective function. These threshold values are the decision variables. Controlling a queue using only one decision variable is not necessarily optimal. Therefore, in some problems policies which have more than one decision variables are used to find the optimal rules to follow [27].

One example of such an approach is presented by Hersh and Brosh [34]. The authors studied the $M/M/1$ system under general rules specifying that the service channel starts to serve customers only if there are N customers waiting in the queue. As soon as the length of the queue is reduced to v, $v < N$, the channel is closed and no more customers are allowed to enter until the queue builds up to N units. We call the pair (v, N) a *policy*, and the goal of the problem is to find the values for v and N that would minimize the cost function. The total expected cost of strategy (v, N) is:

$$k(v, N) = C_L Z_L(v, N) + C_H Z_H(v, N) + C_S Z_S(v, N)$$

where:
C_L - the cost of a lost customer.
C_H - the cost of "hysteresis" - the combined cost of opening (i.e. the numbers of times the channels are opened) and closing (i.e. the numbers of times the channels are closed) the service channel. These costs may be combined since in the steady-state, the number of openings must equal the number of closings [34].
C_S - the cost of service rendered.
$Z_L(v, N)$ - the expected number of lost customers/unit time.
$Z_H(v, N)$ - the expected number of openings or closings of the service channel/unit time.
$Z_L(v, N)$ - the expected fraction of time that the service channel is open.

3.4. Conclusions

In this chapter, general ideas about queueing theory and an overview of queueing optimization problems have been considered. Queueing models are known to be useful for modelling various real-life problems.

Work presented in the following chapters falls under the category of optimal control of queues having a variable number of servers.

Chapter 4

Solving a Queueing Design and Control Problem with Constraint Programming. LogPSums model

4.1. Berman et al.'s problem

In this chapter we are going to present a specific queueing control problem and we are going to investigate the applicability of constraint programming to this problem.

Many retail service facilities are concerned with queueing theory problems, particularly ones which involve optimization. The authors of the paper [37] analyze a problem with front room and back room operations and cross-trained workers. The cross-trained workers are workers who can be switched between the front room and the back room depending on the size of the queue in the front room and the amount of work that has to be completed in the back room. This is a simplified model for such service facilities as banks or stores. In the front room, workers serve the customers who arrive to the facility. The process of arrivals is stochastic, and we assume that the probability distribution describing the times between successive customer arrivals is exponential.

While all workers are busy, customers start to form a queue and have to wait to be served. We assume that since joining the queue, the customers stay until being served. The queue discipline is *First Come*, *First Served* ($FCFS$) (see section 3.2). On the other hand, in the back room, we have a fixed amount of work which has to be done. It is assumed that all workers in the facility are cross-trained and that they can perform both types of work equally well. This assumption allows switching all workers between the front room and the back room depending on the queue length in the front room and the amount of work that needs to be performed in the back room.

In this thesis we deal with the problem of finding a switching policy which minimizes the customers' waiting time in the front room subject to the back room constraint

which requires all work in the back room to be completed. This problem can be viewed as an example of a queueing design and control problem which has been described in section 3.3. Berman et al. presented this problem and their approach in the paper [37].

According to Kendall's notation (see section 3.2.1), the queueing system described by Berman et al. (which will further be referred to as "Problem 1" or $P1$) is denoted as: $M/M/x/S$, where S is a capacity of the system and x is a variable which can be $0, 1, \ldots N$, depending on the policy and the queue length being considered. If there are already S customers in the front room, new arrivals will be blocked from joining the queue. In order to avoid turning customers away from the system, selecting a sufficiently large value for S is necessary. Let λ denote the customer arrival rate and μ the service rate. B_l is the minimum expected number of workers needed in the back room in order to assure that all work is done (back room constraint is satisfied). Only one worker is allowed to be switched from one room to the other at a time.

A switching policy defines the number of workers assigned to the front room depending on the number of customers waiting in the queue and the amount of work in the back room. A policy defined by Berman et al. is described in terms of quantities k_i, $i = 0, \ldots, N$, which have the interpretation that whenever there are between $k_{i-1} + 1$ and k_i customers in the queue, there should be i workers in the front room, $i = 1, \ldots, N$. The goal is to find a switching policy which minimizes the expected customer waiting time (W_q) and satisfies the back room constraint. The switching points k_i are the decision variables and finding the optimal switching policy means finding such an assignment of values to these switching points k_i, $i = 0, \ldots, N-1$ that the *back room constraint* is satisfied and the expected customer waiting time W_q is minimized.

Example 4.1.1. Consider a system with $N = 3$ and $S = 6$, and the policy $(0, 3, 5, 6)$. In this type of policy $k_N = S$, which means that all N workers are employed in the front room when the system reaches its capacity S. This particular policy says that whenever there are between 1 and 3 customers in the front room, there is 1 worker in the front room. When another customer arrives, one more worker is switched from the back room to serve customers in the front room, so when there are 4 and 5 customers, 2 workers are working in the front room. When there are 6 customers, all 3 workers are needed to perform work in the front room.

In the work [39], three different constraint programming models were created to find the optimal values of the switching points k_i.

In the original paper [37], cost and time related to the switches is assumed to be negligible. In this section we are still working under the assumption that switching time and cost are negligible. In the another work of Berman [38], the switching time of workers switched from the front room to the back room is incorporated into the problem. As an extension of the work described in [39], we made an attempt to incorporate this switching time into the problem. This is described in Section 5.1.1 and is based on the idea presented in [38].

4.1.1. Model

Using Berman's notation, define $P(j)$ as the steady-state probability of having j customers in the system, $j = k_0, k_0+1, \ldots, k_N$, where k_0 is a non-negative integer.

The following formulas have been derived by Berman et al. in [37].

In order to obtain the formula for $P(j)$, a set of balance equations $(E1 - EN)$ and the equation:

$$\sum_{j=k_0}^{k_N} P(j) = 1$$

have to be stated.

The following balance equations show the relationship between $P(j)$ and $P(j+1)$ for $j = k_0, \ldots, k_N$:

$$\begin{array}{lll}
(E1) & P(j)\lambda = P(j+1)\mu, & j = k_0, k_0+1, \ldots, k_1-1, \\
(E2) & P(j)\lambda = P(j+2)2\mu, & j = k_1, k_1+1, \ldots, k_2-1, \\
\vdots & \vdots & \vdots \\
(Ei) & P(j)\lambda = P(j+1)i\mu, & j = k_{i-1}, k_{i-1}+1, \ldots, k_i-1, \\
\vdots & \vdots & \vdots \\
(E\mathrm{N}) & P(j)\lambda = P(j+1)N\mu, & j = k_{N-1}, k_{N-1}+1, \ldots, k_N-1.
\end{array}$$

From the balance equations, the following equation has been obtained by authors of [37]:

$$P(j) = \big(\frac{\lambda}{n\mu}\big)^{j-k_{n-1}} \big(\frac{\lambda}{(n-1)\mu}\big)^{k_{n-1}-k_{n-2}} \ldots \big(\frac{\lambda}{\mu}\big)^{k_1-k_0} P(k_0),$$

where: $j = k_{n-1}+1, \ldots, k_n$, $n = 1, \ldots, N$.

$P(j)$ can also be expressed in terms of the probability $P(k_0)$:

$$P(j) = \beta_j P(k_0)$$

where:

$$\beta_j = \begin{cases} 1 & j = k_0 \\ \big(\frac{\lambda}{\mu}\big)^{j-k_0} \big(\frac{1}{n}\big)^{j-k_{n-1}} X_n; \quad k_{n-1}+1 \le j \le k_n & n = 1, \ldots, N \end{cases}$$

and

$$X_n = \prod_{g=1}^{n-1} \big(\frac{1}{g}\big)^{k_g - k_{g-1}}, \quad (X_1 \equiv 1), \quad n = 1, \ldots, N+1.$$

An expression for the expected number of workers in the front room (F) can be expressed in terms of the probability $P(j)$:

$$F = \sum_{n=1}^{N} \sum_{j=k_{n-1}+1}^{k_n} nP(j). \tag{4.1}$$

Therefore, the expected number of workers in the back room can be simply expressed as:

$$B = N - F. \tag{4.2}$$

The expected number of customers in the front room is:

$$L = k_0 P(k_0) + \sum_{j=k_0+1}^{k_N} jP(j), \tag{4.3}$$

and the customer waiting time is:

$$W_q = \frac{L}{\lambda(1 - P(k_N))} - \frac{1}{\mu}. \tag{4.4}$$

The formula for W_q is derived from Little's Laws (see section 3.2.2). Let $\mathbf{K}$ denote the family of all possible switching point policies [37]:

$$\mathbf{K} = \{K; K = \{k_0, k_1, \ldots, k_{N-1}, S\}\},$$

where k_i are integers, $k_i - k_{i-1} \geq 1, k_0 \geq 0, k_{N-1} < S$.

The problem $(P1)$ can be stated as:

$$\begin{gathered} minimize_{K \in \mathrm{K}}\, W_q \\ s.t.\ B \geq B_l \\ equations\ \ (E1)\ to\ (EN) \\ equations\ \ (4.1)\ to\ (4.4) \\ \textstyle\sum_{j=k_0}^{k_N} P(j) = 1 \end{gathered} \tag{4.5}$$

B, F and L are expected values, and therefore can be real-values. The number of workers in the back room at any point in time can be less than B_l at any time because the constraint requires B_l workers to be present in the back room on average, not for every point in time.

4.1.2. A constraint programming model for a queue control problem

In the work [37], the problem $P1$ has been solved by means of heuristic methods.

In the paper [39], a successful attempt of applying constraint programming for the first time to this queue control problem has been described. Authors of [39] created several constraint programming models for the problem stated in [37]. It was found that the most efficient model was the one called $PSums$. The $PSums$ model is based on closed form expressions for the sums of probabilities between two switching points.

$\forall i \in \{1, 2, \ldots, N-1\}$:

$$PSums(k_i) = \begin{cases} P(k_i)\frac{1-\left[\frac{\lambda}{(i+1)\mu}\right]^{k_{i+1}-k_i}}{1-\frac{\lambda}{(i+1)\mu}} & if\ \ \frac{\lambda}{(i+1)\mu} \neq 1, \\ P(k_i)(k_{i+1} - k_i) & otherwise. \end{cases}$$

In other words, the set of probability variables $P(j)$ from the initial formulation described in [37] has been replaced by two sets of variables: variables $PSums(k_i)$, $i = 0, \ldots, N-1$ and probabilities $P(j)$ for $j = k_0, k_1, k_2, \ldots, k_N$. In this case, $P(j)$ is defined only for values $\{k_0, k_1, \ldots, k_S\}$ and not for all j. The variables $PSums(k_i)$ determine the sum of all probabilities between k_i and $k_{i+1} - 1$, $i = 0, \ldots, N-1$.

The constraint programming techniques used for solving this problem performed fairly well. After combining pure CP methods (constraint propagation, search) with other constraint programming procedures, especially with shaving (see section 1.3.2), *PSums* model proved optimality in 79.3% of instances that authors experimented with in [40]. When comparing with other models considered in [39] and [40], *PSums* model found the best-known solution in 97.6% of all instances that were examined. ILOG Solver 6.2, a constraint programming solver, has been used to solve the problem.

Although the constraint programming techniques presented in [40] perform well, none of these methods proved optimality for all tested instances. This is the reason for further investigation of methods for improving the existing models.

In particular, authors of [39] suspected that the decision variables placed in exponents increase the complexity of some expressions of the *PSums* model. This may be one of the reasons for lack of back-propagation in the models presented in [40]. "Back-propagation refers to the pruning of the domains of the decision variables due to the addition of a constraint on the objective function. The objective constraint propagates "back" to the decision variables, removing domain values and so reducing search" [40]. Experimental results proved that even if a constraint is added on the objective function, it has a very small impact on the domains of switching points. Therefore, the idea of creating a new model that is based on the logarithms of the probabilities has been considered in this thesis. Improvement of back-propagation in the model can lead to finding better solutions in a shorter period of time. We examine this idea, create a new model (based on the *PSums* model) and run experiments in order to verify the proposed approach.

4.1.3. *LogPSums* model

As we can see in the *PSums* model, the decision variables are placed in the exponent in formulas for $PSums(k_i)$ for $i = 1, \ldots, N-1$, $P(k_{i+1})$ for $i = 0, \ldots, N-1$ and formulas for β_j for $j = 0, \ldots, S$. In order to get rid of the exponents, we use the logarithmic function and its general properties to rewrite some of the constraints used in the *PSums* model.

Constraints expressed in the *LogPSums*:
$\forall i \in \{0, \ldots, N-1\}$:

$$Log(PSums(k_i)) =$$

$$= \begin{cases} Log(k_i) + Log(1 - (\frac{\lambda}{(i+1)\mu})^{k_{i+1}-k_i}) - Log(1 - (\frac{\lambda}{(i+1)\mu})) & if \quad \frac{\lambda}{(i+1)\mu} < 1 \\ Log(k_i) + Log(1 - (\frac{\lambda}{(i+1)\mu})^{k_{i+1}-k_i}) - Log(1 - \frac{\lambda}{(i+1)\mu}) & if \quad \frac{\lambda}{(i+1)\mu} > 1 \\ Log(k_{i+1} - k_i) + Log(P(k_i)) & otherwise \end{cases} \tag{4.7}$$

$$Log(k_{i+1}) = (k_{i+1} - k_i)(Log(\lambda) - Log((i+1)\mu)) + Log(k_i) \tag{4.8}$$

Constraints involving X and the sum over β_j can be written as:
$\forall i \in \{1, \ldots, N\}$

$$Log(X(i)) = \begin{cases} 0 & for \quad i = 0, 1, 2 \\ Log(X(n-1)) - (k_{n-1} - k_{n-2})Log(n-1) & for \quad i = 3, \ldots, N \end{cases} \tag{4.9}$$

$$\beta Sum(k_i) = \begin{cases} X_i \left(\frac{\lambda}{\mu}\right)^{k_{i-1}-k_0+1} \left(\frac{1}{i}\right) \left[\frac{1-(\frac{\lambda}{i\mu})^{k_i-k_{i-1}}}{1-(\frac{\lambda}{i\mu})}\right] & if \quad \frac{\lambda}{i\mu} \neq 1 \\ X_i \left(\frac{\lambda}{\mu}\right)^{k_{i-1}-k_0+1} \left(\frac{1}{i}\right)(k_i - k_{i-1}) & otherwise \end{cases} \tag{4.10}$$

$$Log(\beta Sum(k_i)) =$$

$$= \begin{cases} if \ \frac{\lambda}{i\mu} < 1 \\ LogX_i + (k_{i-1} - k_0 + 1)[Log(\lambda) - Log(\mu)] - Log(i) + \\ +Log[(i\mu)^{k_i-k_{i-1}} - \lambda^{k_i-k_{i-1}}] - Log(i\mu - \lambda) - (k_i - k_{i-1} - 1)Log(i\mu) \\ \\ \quad if \ \frac{\lambda}{i\mu} > 1 \\ LogX_i + (k_{i-1} - k_0 + 1)[Log(\lambda) - Log(\mu)] - Log(i) + \\ +Log\left[\frac{(i\mu)^{k_i-k_{i-1}} - \lambda^{k_i-k_{i-1}}}{i\mu - \lambda}\right] - (k_i - k_{i-1} - 1)Log(i\mu) \\ \\ \quad if \ \frac{\lambda}{i\mu} = 1 \\ LogX_i + (k_{i-1} - k_0 + 1)(Log(\lambda) - Log(\mu)) - Log(i) + Log(k_i - k_{i-1}) \end{cases}$$

(4.11)

$$Log(P(k_{i+1})) = (k_{i+1} - k_i)Log\lambda - Log((i+1)\mu) + Log(P(k_i)) \tag{4.12}$$

Our new model consists of the following relations and constraints (in remainder of this thesis we will refer to this model as the *LogPSums* model):

$$
\begin{aligned}
&minimize && W_q \\
&subject\ to && \\
&equations\ (E1) && to \quad (EN) \\
&equations\ (4.1-4) && and \quad (4.6-9) \\
&k_i && < \quad k_{i+1};\ \ \forall i \in \{0,1,\ldots,N-1\} \\
&k_N && = \quad S \\
&\sum_{i=0}^{N-1} PSums(k_i) + P(k_N) && = \quad 1 \\
&P(k_0) \times \sum_{i=0}^{N} \beta Sum(k_i) && = \quad 1
\end{aligned}
\tag{4.13}
$$

$$F = \sum_{i=1}^{N} i[PSums(k_{i-1}) - P(k_{i-1}) + P(k_i)]$$

$$L = \sum_{i=1}^{N-1} L(k_i) + k_N P(k_N)$$

$$L(k_i) = k_i PSums(k_i) + P(k_i)\frac{\lambda}{(i+1)\mu}\times$$

$$\times \frac{\left(\frac{\lambda}{(i+1)\mu}\right)^{k_{i+1}-k_i-1}(k_i - k_{i+1}) + \left(\frac{\lambda}{(i+1)\mu}\right)^{k_{i+1}-k_i}(k_{i+1} - k_i - 1) + 1}{\left(1 - \frac{\lambda}{(i+1)\mu}\right)^2}; \forall i \in 0,1,\ldots,N-1$$

$$
\begin{aligned}
W_q &= \frac{L}{\lambda(1 - P(k_N))} - \frac{1}{\mu} \\
N - F &\geq B_l
\end{aligned}
$$

4.2. Experimental Results. *PSums* model vs *LogPSums* model

Our goal is to verify the effectiveness of the model described above and compare the results with the results obtained in [40].

We tested our *LogPSums* model for 300 instances for which the policy described as $\hat{\hat{K}}$ is feasible and policy $\hat{K}$ is infeasible at the same time (see section 4.2.1). However, none of these two policies is optimal. The same instances have been used for *PSums* model.

As input, the program takes 6 different parameters: S, N, λ, μ, Bl and a parameter indicating if we want the program to use the shaving procedures or not. In the previous work [40], several experiments with different shaving procedures and dominance rules (see 1.3.1) incorporated into the solving techniques have been run in order to verify which combination of the procedures is the best. Experimental results showed that

the *PSums* model with the *AlternatingSearchAndShaving* procedure results in the best run time when proving optimality. Here, we run two experiments: one with the shaving procedure, *AlternatingSearchAndShaving*, and the other one just using the standard search procedure.

4.2.1. $\hat{K}$ and $\hat{\hat{K}}$ policies

Two policies have been proposed in [37]:

$$\hat{K} = \{k_0 = 0, k_1 = 1, k_2 = 2, \ldots, k_{N-1} = N-1, k_N = S\}$$

and

$$\hat{\hat{K}} = \{k_0 = S-N, k_1 = S-N+1, \ldots, k_{N-1} = S-1, k_N = S\}.$$

It is stated by Berman et al. that policy $\hat{K}$ yields the smallest possible W_q and B, and the largest possible F. This happens because this policy brings new workers to the front room earlier than any other policy. This implies the following:

$$W_q(\hat{K}) = min_{K \in \mathbf{K}} W_q(K),$$

$$F(\hat{K}) = max_{K \in \mathbf{K}} F(K)$$

and

$$B(\hat{K}) = min_{K \in \mathbf{K}} B(K).$$

On the other hand, policy $\hat{\hat{K}}$ yields the smallest possible F and the largest possible W_q and B.

The problem (P1) is feasible for a set of instances if the back room constraint ($B \geq B_l$) is satisfied for such instances. The policy $\hat{\hat{K}}$ yields the largest possible W_q, if (P1) is not feasible for this policy, there is no other policy for which this problem would be feasible. This happens because the larger value of W_q increases the possibility of satisfaction of the back room constraint.

The facts stated above were both used when implementing all the constraint programming models for the considered problem and when the 300 instances, used in the experiments, were generated. Firstly, the program checks feasibility of the policy $\hat{\hat{K}}$. If the policy is infeasible, the program stops as there is no feasible solution [40]. If the policy $\hat{\hat{K}}$ is feasible, feasibility of the policy $\hat{K}$ is checked. If the policy $\hat{K}$ is also feasible for the examined instances, it is optimal and it is trivial to prove optimality. The experiments have been run on 300 different instances. For these instances, the optimal policy is between $\hat{K}$ and $\hat{\hat{K}}$. In other words, the policy $\hat{\hat{K}}$ is feasible for these sets of instances and the policy $\hat{K}$ is infeasible.

4.2.2. *LogPSums* model with no shaving procedures

In this section we present analysis of the experimental results for the *LogPSums* model without the shaving procedures. Performances of both constraint programming models, the *PSums* model and the *LogPSums* model, are compared.

Number of Times Optimality is Proven

- The *LogPSums* model has proven optimality in 124 cases, which is, 41.3% of all tested instances. The *PSums* model has proven optimality for 2 more cases. However, different sets of instances happened to be easier for these two models.

- In 114 of the cases, both models have proven optimality. However, the *LogPSums* model needed on average 23.94 CPU-seconds more than the *PSums* model (for these particular 114 cases).

- We noticed that the *PSums* model has encountered more difficulty with solving the instances for which $S = 30$ than for any other values for this parameter. For $S = 30$, it has proven optimality just for 7 cases (out of 30). The *LogPSums* model has proven optimality for 12 cases quite quickly. The remaining 13 instances with $S = 30$ happened to be hard to solve for both of the models.

- Both models proved optimality for all 60 instances for which $S = 10$ and $S = 20$; however, it took much more time for the *LogPSums* model to solve these instances to optimality. There were 33 cases for which the *LogPSums* model was not able to find as good a solution as the *PSums* model in the 600 CPU-seconds limit.

- Definitely, the *PSums* model is the best model within all introduced constraint programming models.
  The *PSums* model found the best known solution in 293 cases and the *LogPSums* model found the best known solution in just 267 cases.

Model	S=10	S=20	S=30	S=40	S=50	S=60	S=70	S=80	S=90	S=100
PSums	30	30	7	21	12	14	12	0	0	0
LogPSums	30	30	12	17	12	13	10	0	0	0

Figure 4.1. Number of instances for which the optimal solution was found and optimality was proved within 600 CPU-seconds. There are 10 sets of instances, each of them having a different value of S. For $S = 10$ and $S = 20$ optimality was proved for all instances. For $S = 80$, $S = 90$ and $S = 100$, optimality was not proved in the limit of 600 CPU-seconds for any of the instances.

- Any of the considered models were not able to prove optimality in 164 cases. Besides the 30 cases for which the *PSums* model was better than the *LogPSums* model, both models have found the same solutions in the limit on run time of 600 CPU-seconds.
  There are 12 cases, among the cases for which the optimality have been proved by the *PSums* model, for which the *LogPSums* was not able to prove the optimality. In 9 out of these 12 cases, *LogPSums* found the optimal solution, but did not prove optimality. In the remaining 3 cases, the *LogPSums* model was not able to find the optimal solution in the run time limit (600 CPU-seconds).

- There are 10 cases for which the *LogPSums* model proved the optimal solutions when the *PSums* model did not, and 12 cases with the opposite situation.
  All 10 cases that are solved to optimality by the *LogPSums* model and unsolved by the *PSums* model are instances with S equal to 30 ($S = 30$). In general, the instances with $S = 30$ were hard for the *PSums* model. The *LogPSums* model was able to find solutions for these instances in a significantly smaller amount of time.

Average Run Time

The average run time needed to obtain the optimal solution was 69.92 and 72.75 CPU-seconds for the *PSums* model and for the *LogPSums* model, respectively.

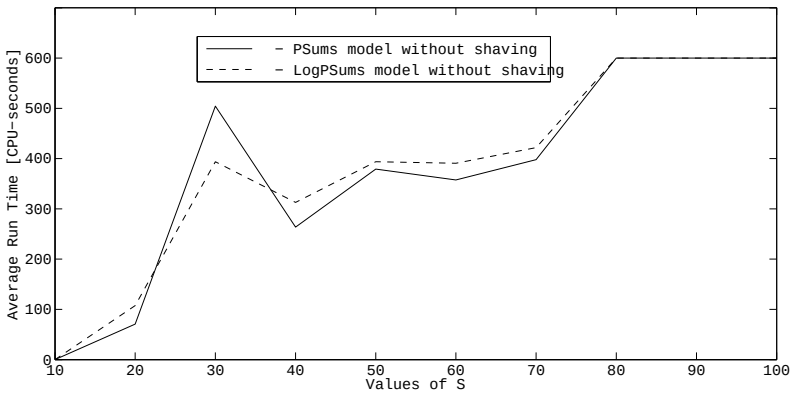

Figure 4.2. Average Run Time needed to find the optimal solution by the compared models without shaving for different S values.

In figure 4.2 we show the average run times for finding the optimal solutions. It is noticeable that parameter S has big impact on the run time. The instances with small S ($S = 10$ and $S = 20$) happened to be the easiest ones for both models. Bigger values of the parameter S make the problem harder to solve.

In the figures below, 4.3 and 4.4, the standard deviations of the run times for different values of S have been shown. Standard deviation is a good measure of the data's dispersion around the mean values.

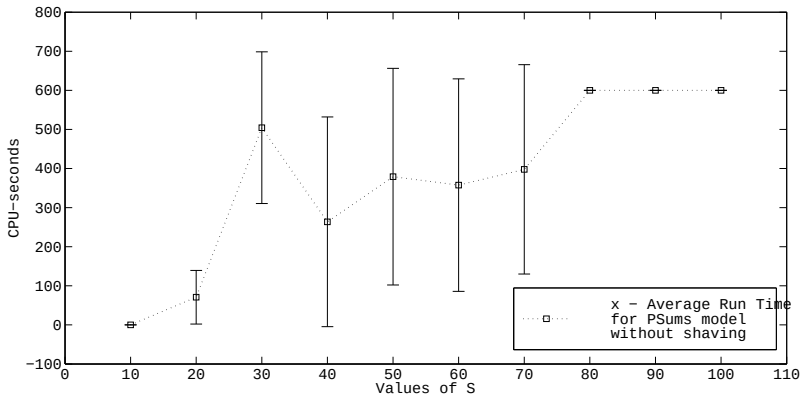

Figure 4.3. The *PSums* model; x - averages of run times for different values of S and standard deviation.

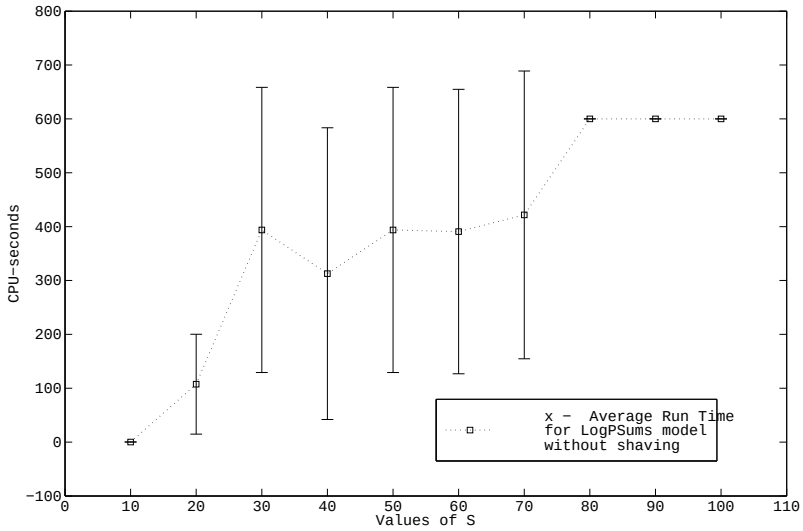

Figure 4.4. The *LogPSums* model; x - averages of run times for different values of S and standard deviation.

Choice Points

In the experiment, we also calculated the number of choice points, that is, the number of nodes in the search tree. We calculated the average number of nodes visited in the search tree before the first feasible solution is found (in the tables below, we refer to them using the title "Before first solution") and the average number of nodes visited within 600-CPU seconds (in the tables below, we refer to them using the title "Total"). In order to compare the two discussed models, these average numbers of visited nodes are calculated for 114 instances solved by both models, 10 instances solved just by the *LogPSums* model and 12 instances solved only by the *PSums* model. A table below shows the results.

	114 Instances solved by All Models	
Model	Before First Solution	Total
PSums	1307	41615
LogPSums	756	41443

Figure 4.5. Average number of explored nodes in the search tree for 114 instances solved by both *PSums* and *LogPSums* model.

	10 Instances Solved by *LogPSums* only		12 Instances Solved by *PSums* Only	
Model	Before First Solution	Total	Before First Solution	Total
PSums	7430	457333	8122	405528
LogPSums	2740	105286	9170	280748

Figure 4.6. Average number of nodes explored in the search tree for both models for 10 cases solved only by the *LogPSums* model and for 12 cases solved only by the *PSums* model

Figure 4.6 shows that for the 10 instances for which only the *LogPSums* model was able to prove optimality, the search tree for the *LogPSums* model is 3 times smaller than the search tree for the *PSums* model.

This happens before finding the initial solutions as well as within 600 CPU-seconds. This means that more propagation is occurring during search in the *LogPSums* model than in the *PSums* model.

4.2.3. Shaving procedures

In the second part of the experiment, we incorporated a shaving procedure (see section 1.3.2) into the program in order to enhance propagation. For the purpose of examining the performance of the *LogPSums* model with the incorporated shaving procedures, we had to add two constraints (part of the *PSums* model) to the initial *LogPSums* model. They constrain β and X_j. Without these constraints, we were not able to receive proper results, since the model could not assign a unique value to each decision variable.

There are two different shaving procedures embedded into the program: Bl-shaving and Wq-shaving.

In the Bl-shaving procedures we consider two cases:

Case 1

$$k_i = \begin{cases} min(k_i) \quad for \quad i = n \\ \\ max(k_i) \quad for \quad i \in \{0, \ldots, N\} \setminus \{n\} \end{cases}$$

The function $gMax$ assigns these values to the switching point variables, and returns true if this assignment results in a feasible policy. If the policy is infeasible, $gMax$ returns false. The assignment is infeasible if the B_l constraint is not satisfied, which happens if $B < B_l$. The condition $k_n < k_{n+1}$, $\forall n \in \{0, \ldots, N-1\}$ must be satisfied when values are assigned to the switching point variables using the function $gMax$.

Actually, the function $gMax$ does not necessarily assign maximal possible values to the variables but rather an upper bound of values that satisfies the condition $k_n < k_{n+1}$ $\forall n \in \{0, \ldots, N-1\}$. In the example described above, if we decide to assign $k_2 = 4$, the function $gMax$ would assign the rest of the values as follows: $k_0 = 2, k_1 = 3, k_3 = 6$. If this policy is infeasible and $min(k_i) + 1 \leq max(k_i)$, a constraint $k_i > min(k_i)$ is temporarily added to the model as increasing the value of switching points will increase the value of B. We know that in this case in any feasible policy k_i must be greater than $min(k_i)$.

At the same time, if the solution is feasible and its W_q is smaller than the W_q for the previous best policy, it is recorded as the best-so-far solution.

Case 2

In the second case, we have exactly the reverse situation. The function $gMin$ is used, which assigns the minimum possible values (lower bounds of possible values for switching point variables) to the variables, except for one switching point for which the maximum value is assigned. All other steps in this shaving procedure are the opposite of the ones described above.

In case 2, the considered policies result in a smaller waiting time than the ones considered in case 1. Therefore, our procedure starts with $gMin$ - the function described in case 2. One can find the exact description of the algorithms used in the experiment in [40].

W_q-based shaving

The W_q-based shaving procedure adds the model the constraint $W_q \leq bestW_q$ to the model. In order to avoid incorrect inferences, the constraint $B \geq B_l$ is removed.

Similarly, as described above, a constraint $k_i = max(k_i)$ is added and the function $gMin$ is used in this procedure. In this procedure, a policy is considered to be infeasible only if the constraint $W_q \geq bestW_q$ has been violated.

In general, the technique which has been applied to our problem consists of two alternating procedures: W_q-based shaving and B_l-based shaving.

The W_q-based shaving procedure reduces the upper bounds of some variables. After successfully performing W_q-based shaving, B_l-based shaving is applied and a new set of policies is generated and considered by the function $gMax$. This procedure again leads to domain reductions. If the B_l-based shaving procedure finds a new best solution, then W_q-based shaving may be able to make new domain reductions by applying the new value of $bestW_q$. These two procedures are applied until no more domain reductions are possible.

In the *LogPSums* model, when no shaving was applied, the program was not able to assign a unique value to the variables in 27 cases.

We use an example from Berman's work [37] as an illustration of the procedures discussed above.

Example 4.2.1. For the instance:
$N = 3$, $S = 6$, λ=15, μ=3, B_l=0.32, the policy $\hat{K}$ is

$$(k_0, k_1, k_2, k_3) = (0, 1, 2, 6)$$

and the policy $\hat{\hat{K}}$ is

$$(k_0, k_1, k_2, k_3) = (3, 4, 5, 6).$$

As explained in the section 4.2.1, policy $\hat{K}$ yields the smallest possible W_q and policy $\hat{\hat{K}}$ yields the greatest possible W_q. The initial domains of switching points are therefore: $[0 \ldots 3]$, $[1 \ldots 4]$, $[2 \ldots 5]$ and $[6]$ for k_0, k_1, k_2 and k_3 respectively. Shaving procedures may be able to reduce these domains at any step.

4.2.4. *LogPSums* model with shaving

Number of Times Optimality Proven

- The *LogPSums* model proved optimality in 223 cases, which is 74.3% of all tested instances. The *PSums* model proved optimality for 15 more cases. All 223 cases solved by the *LogPSums* model have also been solved by the *PSums* model. Therefore, both models proved optimality in 223 cases.
- The *LogPSums* model required 156 CPU-seconds on average in order to solve an instance of the problem. The *PSums* model happened to be faster, requiring 130 CPU-seconds on average.
- For both models, instances with parameter $S = 30$ happened to be relatively hard. In general, instances with smaller values of S are easier to solve. Both models proved optimality in all instances with the parameter S equal to 10 and 20. However, this took more time for the *LogPSums* model than for the *PSums* model.

- In table 4.7, we compare the number of instances for different values of S for which optimality has been proven.

Model	S=10	S=20	S=30	S=40	S=50	S=60	S=70	S=80	S=90	S=100
PSums	30	30	23	30	27	24	22	17	20	15
LogPSums	30	30	21	28	26	22	22	15	16	13

Figure 4.7. Number of instances for which the optimal solution was found and its optimality was proved within 600 CPU-seconds run time limit. For $S = 10$ and $S = 20$, optimality was proven for all 60 instances. For $S = 40$, optimality was proven by the *PSums* model in all 30 cases, but the *LogPSums* model proved optimality in 28 cases.

- In 62 cases, neither of the models was able to prove optimality in the limit of 600 CPU-seconds. However, the best solutions (value of W_q) for these cases were the same for both models.

Average Run Times

The *LogPSums* model needed 8.06 CPU-seconds on average in order to find and prove the optimal solution. The average run time for the *PSums* model is equal to 7.62 CPU-seconds. These numbers are arithmetic means of the run times for the instances for which optimality was proved, i.e. 223 and 238 cases for *LogPSums* model and for *PSums* model respectively.

Taking into account all 300 instances, the average time needed by the *LogPSums* model in order to solve the problem was 156 CPU-seconds. For the *PSums* model, the average run time was 130 CPU-seconds.

The general observation is that the *PSums* model is faster than the *LogPSums* model. Figure 4.8 shows the average run time for different values of S for both models with shaving.

The graphs 4.9 and 4.10 show the standard deviation.

It is easy to notice a big spread around the mean values of the run times. This observation suggests that the difficulty of solving the problem does not depend only on the value of S. In both *PSums* and *LogPSums*, there were hard and simple cases for each set of 30 instances for different values of S.

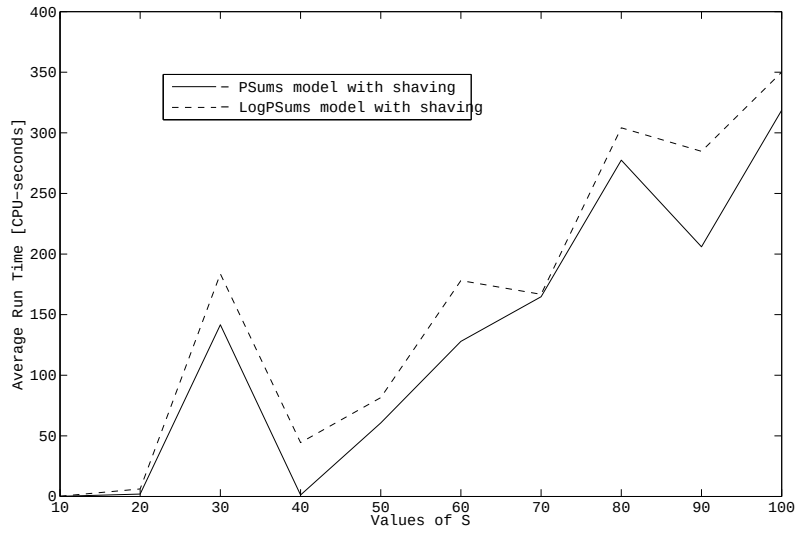

Figure 4.8. Average Run Time needed to find the optimal solution by the compared models with shaving for different S values.

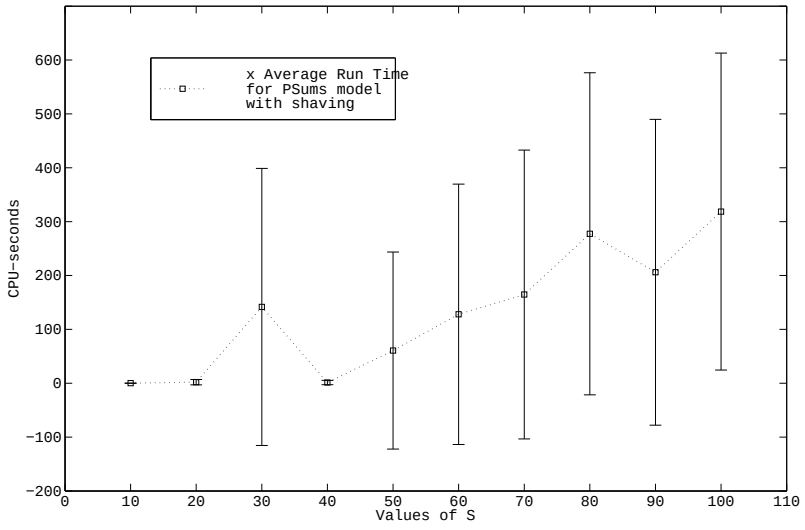

Figure 4.9. *PSums* model with shaving; x - averages of run times for different values of S and standard deviations.

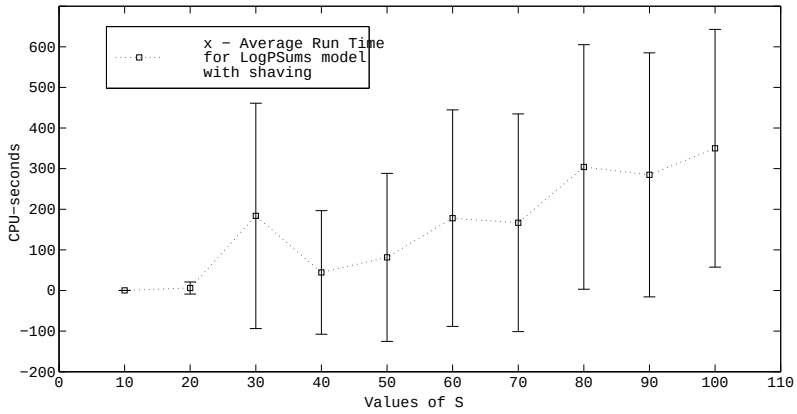

Figure 4.10. *LogPSums* model with shaving; x - averages of run times for different values of S and standard deviations.

Chapter 5

Switching time

5.1. Discussion of switching time and switching cost

Besides the impact on the customer waiting time, other consequences of switching the workers between the back room and the front room have not been analyzed in the previous chapter. However, there are several other aspects that could be considered. When a worker is being switched from one room to another, he has to move. In some cases this distance is very small, so it can be neglected. However, there are situations when the distance is significant, and the time needed for moving from one place to another has to be considered when the decision about switching is being made. Another consequence arising from switching is lost productivity by the switched workers. This happens because a switched worker has to acclimatize to performing new work- he has to find out what the next step is, etc. We will refer to the time that a worker needs for moving from one place to another and to readjust to new work as *switching time*. Furthermore, no work is being performed by the worker while the switch is happening. Therefore, there is a cost incurred by the company due to this fact. We will refer to this cost as *switching cost*.

This aspect of the problem was examined in a paper by Berman and Larson [38].

As an extension of the work described in [39], we made an attempt to incorporate the *switching time* into the problem based on the work of [38].

The problem examined by Berman and Larson [38] is different from the one examined by Berman, Wang and Sapna [37]. In the paper [38], there is an assumption stating that "work in the back room is interruptible, allowing a worker in the back room to switch to front room work with little delay or lost productivity, while switching from the front room to the back room (possibly) requires some $re-orientation\ time$ (for that particular worker)". This comes from the fact that the worker switched to

work in the front room has to become familiar with the status of the work and with the next step that needs to be taken. However, the paper [37] neglects this fact, allowing workers to switch between the front room and the back room without losing time or productivity.

Another difference in the problem definition is that, in [38], there is a distinction between the customers who are in the store (customers who are shopping) and the customers who are at the checkout. Berman et al. [37] do not make this distinction. Moreover, the decision to switch workers between rooms in [38] depends not only on the number of customers in the queue but also on the overall number of customers in the store. In [37], only the number of customers in the queue is taken into account.

In this chapter, the cost arising from these switches has been taken into account. The time which the worker loses for the readjustment is assumed to be a fraction of the hour τ ($\tau \geq 0$) [38]. As stated in [41], this aspect of switching is important because the switches that take a long time may result in a greater customer waiting time, W_q, than in the model which does not consider switching time.

Therefore, one of our aims is to examine the impact of this modified assumption on the waiting time (W_q) and find out if ignoring this aspect of the problem brings about a big difference.

5.1.1. Switching from the front room to the back room

We assume that each time when a worker is being switched to the back room, he loses τ units of time (fraction of an hour) in order to acclimatize. Therefore, the actual expected number of workers in the back room is reduced by $\nu_{back}\tau$, where ν_{back} is the frequency of workers returning to the back room per hour. We may need to take into account the travel time that the worker loses during each switch when he is moving between the two rooms. We assume that this fraction of the hour is included in τ.

Following the authors of [38], we denote this effective time-average worker complement in the back room as B_e, and

$$B_e = B - \nu_{back}\tau.$$

In the initial $PSums$ model the constraint that assured having enough workers in the back room to complete the required work was stated as: $B \geq Bl$. In order to incorporate switching time, the above constraint is stated as:

$$B - \nu_{back}\tau \geq Bl.$$

This constraint is more restrictive, since it demands a reduced value of B to still be greater than Bl.

5.1.2. Switching from the back room to the front room

Let τ_{front} denote the switching time needed to re-adjust to working in the front room. Therefore, we can calculate the average number of workers, who, in spite of being assigned to work in the front room, are not actually working because they are in the state of preparing themselves to start working. This average number of workers is equal to $\nu_{front}\tau_{front}$.

Since 1 worker serves μ customers during one unit of time, by using a simple proportion we find that $\nu_{front}\tau_{front}$ workers would serve $\nu_{front}\tau_{front}\mu$ customers during one unit of time. This influences the number of customers in the front room which is now equal to:

$$L' = L + \nu_{front}\tau_{front}\mu.$$

Switching time also has a direct impact on the customer waiting time: $W_q = \frac{L'}{\lambda(1-P(k_N))} - \frac{1}{\mu}$. However, the correctness of this formula requires further verification, for example by simulation.

5.1.3. Switching cost

Another way of incorporating lost productivity due to switches between the rooms is to calculate the switching cost. Assuming that each switch implies some fixed cost c, the total cost for switches is equal to $c\nu$. Obviously, we want to minimize this total cost. Therefore, our objective function is now: $f_{switch.cost} = c\nu$. Our second goal is also to minimize W_q - therefore setting an upper bound for W_q is necessary.

The overall model for this extended problem is:

$$min \quad c\nu$$

$$s.t.\ B \geq B_l$$

$$equations \ \ (E1)\ to\ (EN)$$

$$equations \ \ (4.1)\ to\ (4.4)$$

$$\sum_{i=0}^{N-1} PSums(k_i) + P(k_N) = 1$$

$$\nu = \sum_{i=0}^{N-1} (i+1)\mu P(k_i + 1)$$

$$W_q \leq W_u$$

where the upper bound W_u is a problem parameter.

5.2. Model with switching from the front room to the back room

Following the model introduced by Berman and Larson in [38], we considered the switching time as a result of switches from the front room to the back room. In this problem, we assumed that the time lost by a worker when switching from the back room to the front room is relatively small, so we can ignore it. As an explanation of this we could use the fact that in most cases there is always just one type of work that needs to be performed in the front room - serving the customers. Therefore, the worker does not need any meaningful amount of time for acclimatization and orientation to the work which has to be performed. Therefore, there is no impact on the calculation of F - the time-average number of workers in the front room.

In this section we discuss including the switching time into the problem in order to find out how big an impact it has on the customer waiting time and the average number of customers in the front room.

In order to modify the back room constraint, the formula for calculating the frequency of switches, ν_{back}, has to be derived. The desired formula can be determined from the steady state probabilities [38]. As in Chapter 4, the switching policy is defined as a vector of decision variables $(k_0, k_1, \ldots, k_N)$ where $k_i, i \in 0, \ldots, N$ are the switching points. This notation means that each time when there are between $k_{i-1}+1$ and k_i customers in the system, there are i workers in the front room. Hence, each time there are k_i customers in the system and another one arrives, the switch from the back room occurs [39]. On the other hand, each time when there are $k_{i-1}+1$ customers in the system and one gets served and leaves, the switch from the front room to the back room occurs.

According to these facts, the expected frequency of switches is "the product of the long-run proportion of time when the system is in a state when a switch is possible and the rate at which arrivals (or departures) occur when the system is in such a state" [40].

So, the frequency of switches can be calculated as follows:

$$\nu_{back} = \sum_{i=0}^{N-1} (i+1)\mu P(k_i + 1),$$

where μ - the service rate.

The overall model can be presented as follow:

$$min \quad W_q$$
$$s.t. \text{constraints from the PSums model}$$
$$\nu_{back} = \sum_{i=0}^{N-1} (i+1)\mu P(k_i + 1)$$
$$B_e = B - \nu_{back}\tau$$
$$B_e \geq Bl$$

Since both frequencies ν_{back} and ν_{front} should be equal in the long run, we will refer to ν_{back} as a ν from now on.

5.2.1. Experiments

In this section we present results of the experiments performed in order to verify the efficiency of the model described in the previous section. The model was implemented in ILOG Solver 6.2. Since $PSums$ was determined to be the best constraint programming model out of the ones proposed in [39], we incorporate switching time into this model.

The constraint that assures that all work in the back room is completed is expressed in our model as:

$$B_e = B - \nu_{back}\tau_{back} \geq B_l. \tag{5.1}$$

Note that in this expression we subtract $\nu_{back}\tau_{back}$, the time-average number of workers who are in the process of switching, from the expected number of workers in the back room (B).

If the policy $\hat{\hat{K}}$ is not feasible for some particular instances then the problem is not feasible for these instances [37].

Corollaries

In Berman et al.'s work [37], two very intuitive corollaries have been stated which concern policies $\hat{K}$ and $\hat{\hat{K}}$ (see section 4.2.1).

Corollary 5.2.1. Given N, the policy $\hat{K} = \{k_0 = 0, k_1 = 1, k_2 = 2, \ldots, k_{N-1} = N - 1, k_N = S\}$ yields the smallest possible W_q and B, and the largest possible F, i.e.

$$W_q(\hat{K}) = min_{K \in K} W_q(K),$$

$$B(\hat{K}) = min_{K \in K} B(K),$$

and

$$F(\hat{K}) = max_{K \in K} F(K).$$

Corollary 5.2.2. Given N, the policy $\hat{\hat{K}} = \{k_0 = S - N, k_1 = S - N + 1, k_2 = S - N + 2, \ldots, k_{N-1} = S - 1, k_N = S\}$ yields the smallest possible F and the largest possible W_q and B, i.e.

$$W_q(\hat{\hat{K}}) = max_{K \in K} W_q(K),$$

$$F(\hat{\hat{K}}) = min_{K \in K} F(K),$$

and

$$B(\hat{\hat{K}}) = max_{K \in K} B(K).$$

The policy $\hat{\hat{K}}$ results in the largest possible customer waiting time and the largest expected number of workers in the back room because this policy assumes switching workers from the back room to the front room as late as possible.

Problem ($P1$) concerns the minimization of customer waiting time subject to the Bl constraint $B \geq B_l$. This constraint says that there have to be at least Bl workers in the back room in order to perform the required work. In the class of considered policies, the policy $\hat{\hat{K}}$ yields the largest possible B.

Therefore, the main conclusion obtained from Corollary 5.2.2 is:

Statement 1: "If policy $\hat{\hat{K}}$ is infeasible, problem $P1$ is infeasible".

Experimental Results

In order to examine the impact of the switching time (τ) on the overall results for the $PSums$ model, we run experiments on the same set of instances as in the experiment with the $PSums$ model conducted and described by the authors of the work [41]. We compared the results of the experiments for parameters with $\tau > 0$ and with $\tau = 0$. This helped us to evaluate how the customer waiting time is changing if the switching time is included. We tested the model with the alternating shaving procedures since this model is the most efficient.

We applied Statement 1, quoted above, in order to generate feasible instances for running the experiments. Therefore, experiments were performed for the 300 instances for which policy $\hat{\hat{K}}$ is feasible and policy $\hat{K}$ is not feasible.

In order to generate instances for the experiments, we first had to find adequate values for τ that can allow for a good approximation of real life situations. We made an attempt of assigning a higher value for switching time, namely $\tau \in (0.5; 1)$. This range has been proposed in [38]. However, using this range we were not able to obtain any feasible solutions.

The reason why this happened is that as the value for τ is increased, the expected number of workers needed in the back room in order to perform the compulsory work (Bl) is increased as well.

In order to be accurate in choosing values for switching time, we tested several different values from range: $\tau \in (0.1; 0.4)$. In general, switching time should depend on the store, the performed work, and many different conditions. Small values for τ, from the range $(0; 0.1)$ did not give us significant differences from the results obtained for cases when $\tau = 0$.

Other observations are that for relatively big values of arrival and service rates we did not obtain any feasible solutions when including the switching time ($\tau > 0$).

In general, observations resulting from the experiment are that including switching time into the model always increases the mean customer waiting time. Since the efficiency of the service facility and the service quality are the most important aspects of the problem, it is important to take switching time into account. Omitting it does not yield an accurate approximation of the real situation.

We need to mention that for our instances, generated as described above, after assigning zero to the parameter τ, for all 300 instances we obtained trivial instances. This means that the $\hat{K}$ policy is feasible for these instances and therefore optimal. This fact indicates that the parameter τ plays a very important role in the problem and changing its value changes the solution to the problem.

In several cases when including the switching time, the value of the waiting time has been changed significantly. Table 5.1 below presents the biggest differences that we have obtained for several instances.

The biggest difference between the expected waiting times obtained for the instances for which $\tau > 0$ and for the same instances with $\tau = 0$ was observed for $S = 50$. On the other hand, the smallest difference was observed for $S = 30$, we plot these differences on the graph 5.2.

Count	W_q for $\tau = 0.1$	W_q for $\tau = 0$	Difference
1	8.68673	2.22526	6.46147
2	6.5025	$1.25855 * 10^{-5}$	6.502487415
3	6.7403	$9.88823 * 10^{-6}$	6.740290112
4	8.35492	0.222222	8.132698
5	8.68673	2.22526	6.46147
6	8.59387	2.0201	6.57377
7	7.27952	1.2697	6.00982

Figure 5.1. The biggest difference that we have obtained between expected waiting time with switching time ($\tau = 0.1$) and without switching time ($\tau = 0$).

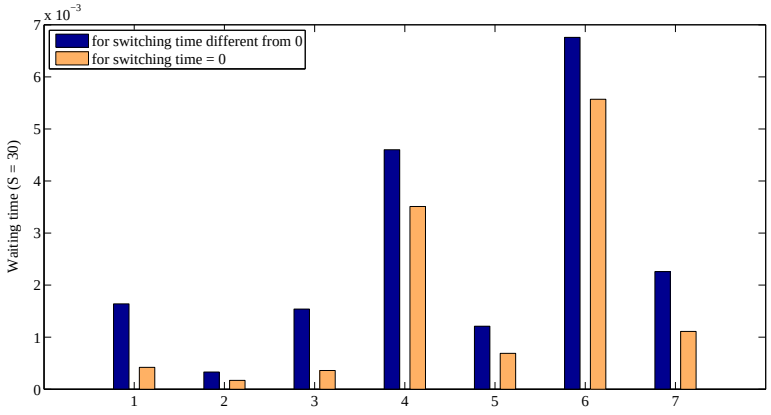

Figure 5.2. Expected waiting time for selected instances for $S = 30$, for which the difference between the W_q values when $\tau > 0$ and when $\tau = 0$ is small compared to the other cases.

5.3. Conclusions

After running the experiments, we can conclude that neglecting the switching time produces inaccurate results. Incorporating it into the problem gives us a more reliable model.

As the results showed, when trying to optimize the operations of service facilities, it is worthwhile to consider the amount of time that the workers are losing during the switch between the different types of work. This is very important, especially for companies where workers perform several different tasks depending on the system state. The manager has to decide when he should switch workers to perform a particular task, and he has to take into account the cost that the company incurs for customer waiting time. In all service retail facilities, efficiency is crucial.

As future work, one could verify the idea presented in section 5.1.2. The problematic issue is incorporating the switching time of workers switching from the back room to the front room into the model. Another idea is to incorporate the switching cost into the problem, as described in the subsection 5.1.3.

Bibliography

[1] D. Gross and C.M. Harris. Fundamentals of Queueing Theory. John Wiley and Sons, Inc. 1998.

[2] R. Dechter. Constraint Processing. Morgan Kaufmann, 2003.

[3] K.R. Apt. Principles of Constraint Programming. Cambridge University Press, 2003.

[4] B.M. Smith. Modelling for Constraint Programming. Cork Constraint Computation Centre, University College Cork, Ireland, 2005.

[5] R. Bartak. Constraint Programming - What is behind? In: J.Figwer (editor) Proceedings of the Workshop on Constraint Programming in Decision and Control, June 1999, Poland.

[6] R. Bartak. Constraint Programming: In Pursuit of the Holy Grail. Charles University, Faculty of Mathematics and Physics, Department of Theoretical Computer Science. Prague, Czech Republic.

[7] T. Walsh. Constraint patterns. In: F.Rossi(Editor), Principles and Practise of Constraint Programming- CP2003, LNCS 2833, Springer 2003, pages 53-64, invited talk.

[8] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. AI Magazine, 1992, pages 33- 44.

[9] A. Moallem. Cost Optimization in Production Line: A Genetic Algorithm Approach. Faculty of Electrical and Computer Eng., School of Engineering, University of Tehran.

[10] T. Fruhwirth and S. Abdennadher. Essentials of Constraint Programming. Springer, 2003.

[11] W. Legierski. Planowanie zajęć metodami programowania z ograniczeniami. In Automatyzacja procesów dyskretnych. Poland.

[12] E. Tsang. Foundations of constraint satisfaction., Essex, UK, 1996.

[13] S.D. Prestwich and J.C. Beck. Exploiting Dominance in Three Symmetric Problems. In Proc. of the 4th International Workshop on Symmetry and Constraint Satisfaction Problems, pages 63-70, 2004.

[14] T. Ibaraki. The Power of Dominance Relations in Branch-and-Bond Algorithms. Journal of the Association for Computing Machinery, vol. 24, 1977, ACM Press, pages 264-279.

[15] T. Walsh. Stochastic constraint programming. In Proceedings of the Fifteenth European Conference on Artificial Intelligence, pages 111–115, 2002.

[16] S. Manandhar, A. Tarim and T. Walsh. Scenario-based stochastic constraint programming. In Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'2003), 2003.

[17] T. Walsh. Stochastic OPL. Cork Constraint Computation Centre, Cork, Ireland, 2002.

[18] T. Balafoutis and K. Stergiou. Algorithms for stochastic CSPs. Principles and Practice of Constraint Programming - CP 2006, vol. 4204, 2006.

[19] A. Tarim, S. Manandhar and T. Walsh. Stochastic constraint programming: A Scenario Based Approach. Constraints, 11(1), 2006.

[20] Wikipedia, The Free Encyclopedia, 2007. Available at: http://en.wikipedia.org.

[21] F.S. Hillier. Economic models for industrial waiting line problems. Management Science, Vol. 10, No. 1, pages 119-130, 1963.

[22] G. Brigham. On a Congestion Problem in an Aircraft Factory. Journal of the Operations Research Society of America, Vol. 3, No. 4, pages 412-428, 1955.

[23] O. Kella. Optimal control of the vacation scheme in an M/G/1 queue. Technical Report, Yale University, New Haven, Connecticut, 1989.

[24] P.M. Morse. Queues, Inventories and Maintenance. New York. Wiley, 1958.

[25] R.V. Evans. Programming problems and changes in the stable behavior of a class of Markov chains. Journal of Applied Probability, Vol. 8, No. 3, pages 543-550, 1971.

[26] J.J. Moder and C.R. Phillips Jr. Queuing with Fixed and Variable Channels. Operations Research, Vol. 10, No. 2, pages 218-231, 1962.

[27] L. Tadj and G. Choudhury. Optimal Design and Control of Queues. Sociedad de Estadística e Investigación Operativa, Top (2005), Vol. 13, No. 2, pages 359-412.

[28] S.Jr. Stidham. On the Optimality of Single-Server Queuing Systems. Operations Research, Vol. 18, No. 4., pages 708-732, 1970.

[29] M. Yadin and P. Naor. Queueing Systems with a Removable Service Station. Operational Research Quarterly, Vol. 14, pages 393-405, 1963.

[30] N. Igaki. Exponential two Server Queue with N-policy and General Vacations. Queueing Systems, Vol. 10, pages 279-294, 1992.

[31] H.W. Lee and K.S. Song. Queue Length Analysis of MAP/G/1 Queue Under D-policy. Stochastic Models, Vol. 20, pages 363-380, 2004.

[32] Ho Woo Lee and Jung Woo Baek. BMAP/G/1 queue under D-policy: queue length analysis. Stochastic Models, Vol. 21, pages 485 - 505, Issue 2 3 January 2005.

[33] J.R. Artalejo. A Note on the Optimality of the N- and D-policies for the M/G/1 Queue. Operations Research Letters 23, pages 35-43, 2002.

[34] M. Hersh and I. Brosh. The Optimal Strategy Structure of an Intermittently Operated Service Channel. European Journal of the Operational Research Society, Vol. 5, pages 133-141, 1980.

[35] A. Ferrier. Kendall's Notation for Classification of Queue Types. Last updated: June 6, 1999. <http://new-destiny.co.uk/andrew/pastwork/queueingtheory/Andy/kendall.html>

[36] J.D.C. Little. A Proof of the Queueing Formula $L = \lambda W$. Operations Research, 9, pages 383-387, 1961.

[37] O. Berman, J. Wang and K.P. Sapna. Optimal management of cross-trained workers in services with negligible switching costs. European Journal of Operational Research, 167 (2), pages 349-369, 2005.

[38] O. Berman and R. C. Larson. A queueing control model for retail services having back room operations and cross-trained workers. Computers and Operations Research, 31 (2), pages 201-222, 2004.

[39] D. Terekhov, J. C. Beck, and K. N. Brown. Solving a stochastic queueing design and control problem with constraint programming. In Proceedings of Twenty-Second National Conference on Artificial Intelligence (AAAI'07), 2007.

[40] D. Terekhov. Solving queueing design and control problems with constraint programming. Master Thesis, Department of Mechanical and Industrial Engineering, University of Toronto, 2007.

[41] D. Terekhov and J. C. Beck. Solving a stochastic queueing control problem with constraint programming. In Proceedings of the Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial

Optimization Problems (CPAIOR'07), pages 303–317. Springer-Verlag, 2007.

[42] J.E. Beasley. OR-Notes. Available at:
http://people.brunel.ac.uk/mastjjb/jeb/or/queue.html