TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

M A S T E R A R B E I T

An Incremental Dynamic Programming Approach
for
Multidimensional Allocation Problems

ausgeführt am Institut für
Computergraphik und Algorithmen
der Technischen Universität Wien

unter der Anleitung von
Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
und
Dip.-Ing. Dr.techn. Alfred Kalliauer (VERBUND - Austrian Power Trading AG) als Mitbetreuer

durch
Christoph Bonitz

Gumpendorfer Straße 83-85/1/20
1060 Wien, Österreich

Datum

Unterschrift

An Incremental Dynamic Programming Algorithm for Multidimensional Allocation Problems

Abstract

This thesis presents an incremental Dynamic Programming (DP) Approach called *Optimal Policy Iteration* for solving an allocation problem inspired by Hydro Storage System Planning. Hydroelectric power plants have the unique property of being able to efficiently store large amounts of energy. To achieve good profits, it is necessary to plan when to use this energy to generate electricity.

The formal problem addressed in this thesis is finding an optimal policy for allocating a good, for example water, represented in discrete units, from several stores with limited capacity over a finite period of time (represented by discrete decision points). The number of allocations can be constrained for each time step as well as for each combination of store and time step. The objective value models the expected development of the good's market price, and is computed as the sum of concave functions of the decision variables called *pricing functions*.

It is shown that, because of the concave pricing functions, two problem instances that are identical with respect to allocation limits and differ only by one unit in the capacity of a single store have optimal solutions that lie within a defined neighborhood of each other. This neighborhood can efficiently be searched by Dynamic Programming.

An algorithm is presented that uses this property to find an optimal policy for a given problem instance using a series of optimal policies for sub-problem-instances that are "growing" with respect to capacities, starting with the optimal solution to a trivial sub-problem. This algorithm, *Optimal Policy Iteration*, is compared with a standard Dynamic Programming approach as well as an *Evolutionary Algorithm*.

The algorithm performs significantly better than a simple DP solution, making it possible to solve problem instances with large capacities. The evolutionary algorithm performs better on instances with many stores, but is not guaranteed to give optimal solutions.

The algorithm uses a deterministic pricing scheme. In its current form it cannot simply be extended to stochastic pricing models. A counterexample shows that the uncertainty about future prices makes it impossible to directly translate the locality results from deterministic pricing to a price scenario.

1 Acknowledgements

There are several people wihout whom this thesis project would not have been possible. I want to express my gratitude to Professor Günther Raidl, the advisor of my thesis project, for support, advice, patience, feedback, corrections of drafts and the possibility of attending the *EURO XXII-Conference*. The algorithm analyzed in this thesis was devised by Alfred Kalliauer, who also proposed this thesis project. Thanks to him for his help, feedback and encouragement during the project. Special thanks go to my parents Eva and Wolfgang Bonitz for encouraging and supporting my studies.

Christoph Bonitz
Vienna, May 2008

Contents

2 Introduction

This chapter discusses the motivation for solving the allocation problem discussed in this thesis. Furthermore, an of the algorithm's history is given.

2.1 Motivation

This thesis thoroughly analyzes an algorithm that solves an allocation problem inspired by resource planning for reservoirs of hydroelectric power plants. Reservoirs of hydroelectric power plants store massive amounts of water behind a dam. When needed, the potential energy of the water can be transformed into electrical energy using turbines attached to generators. This flexibility allows operators to generate electricity when the expected profits are highest.

There are two ways the reservoirs can be filled with water. First, there is the natural inflow of water from the river on which the plant is built. Second, if the operator controls at least two reservoirs near to each other and one is below the other, water can be pumped from the (physically) lower plant to the higher one using electricity when the prices are low, and used to create electricity again when the prices are higher, allowing the operator to take advantage of fluctuations of energy prices Independently of how the reservoir was filled, the water inside it should be used for the generation of electricity when the prices are highest.

In this thesis we consider a simplified model that treats the water in a reservoir as stored electrical energy that can be allocated in discrete units over a given period of time, trying to optimize the profit from selling them, given an expected market price represented by the sum of concave pricing functions. We do not consider inflow, pumping or evaporation losses.

We look at a recursive definition for the objective function value of a problem instance, and describe a simple *Dynamic Programming (DP)* algorithm based on this recursion. However, the running time of the algorithm dramatically increases with growing store capacities.

To solve larger instances, this thesis presents an algorithm that exploits a property guaranteed by the concave pricing functions: When two problem instances differ only by one unit in the capacities of one store and are identical with respect to all other constraints, their optimal solutions are similar. Given the optimal solution to one of them, the other instance's optimal solution can be found with an efficient local *DP* search. The size of the search space only depends on the number of dimensions and time steps.

Given a problem instance I, the algorithm, *Optimal Policy Iteration (OPI)*, starts with a trivial sub-problem-instance I_0 that has the same allocation limits as I but a capacity of zero units in each store. The optimal solution to I_0 is not allocating any units, as there are none. Using the optimal solution to I_0 we can now efficiently find the optimal solution to a problem instance that has one unit in one store, and zero in all the others. This scheme can be continued iteratively, adding one unit per iteration, until we have the optimal solution to our original problem instance. For this algorithm, the store capacities are only a linear factor.

This thesis describes the algorithm in detail. Its correctness is shown in a rigorous formal proof. A description of the implementation is provided. Computational experiments examine the performance of two variants of this approach

and compare it to the simple *DP* approach described above and an *Evolutionary Algorithm*.

2.2 History and Related Work

This thesis is based on work by Alfred and Andrea Kalliauer. In 2002 they discovered a method for evaluating option prices, which they presented at two conferences [11], [12]. It is similar to an independently discovered method of evaluating *Swing Options* [9] published in 2004.

In 2006 it was discovered that this principle could be extended to multi-dimensional allocation problems ([2], [1], [13]). At the end of 2006, Alfred Kalliauer contacted Professor Günther Raidl, head of the *Algorithms and Data Structures Group* of the *Institute of Computer Graphics and Algorithms* at *Vienna University of Technology*. The aim was a thorough analysis as well as an implementation of his algorithm during a thesis project.

This is how this thesis came to be. The project resulted in the first implementation and thorough correctness proof of Alfred Kalliauer's algorithm, and was also presented at the 22^{nd} *European Conference on Operational Research* in 2007 [4].

3 Related Theory

Several chapters of this thesis requires some theoretical background. This chapter gives an overview of the most important theoretical concepts used in this thesis and contains references to related articles and books.

3.1 Optimization Problems

Optimization problems are problems where the aim is to minimize or maximize the value of an *objective function* by assigning values to *decision variables* such that a set of *constraints* are met. That is, from a set of *legal solutions* choose one with maximal or minimal objective function value.

Combinatorial optimization is the class of optimization problems where the number of legal solutions is finite. A well known example is the classic *Knapsack Problem*, where we have to choose a subset a finite set of goods that each have a profit and a weight assigned to them, in a way that maximizes the sum of the profits but keeps the weight under a given limit. Since the number of subsets of a finite set is finite, the number of legal solutions must be finite as well.

Conversely, if the number of legal solutions is unlimited, the problem is called a *continuous optimization problem*. As an example, consider the *fractional Knapsack Problem*, where we are allowed to pick fractions of each good, yielding an infinite number of legal solutions.

3.2 Dynamic Programming

Dynamic Programming or *DP* is a technique used primarily to solve optimization problems. Its theoretical foundations have first been extensively researched by Richard Bellman in the 1950s [3].

The well-known algorithms textbook [5] states two important properties of problems to be solved with *Dynamic Programming*:

- optimal substructure
- shared subproblems

Overlapping subproblems describes problems which are composed of subproblems, which, again, consist of several subproblems, with the property that one sub problem occurs in several (sub)problems. This means that a top-down, recursive approach to solving such a problem would result in subproblems being solved several times. *Optimal substructure* means that each of the subproblems of an optimal solutions need to be solved in an optimal manner as well, i.e. a non-optimally solved sub-problem in an optimal solution would yield a contradiction.

Dynamic Programming is a bottom-up approach. It starts by solving smaller subproblems to optimality and using them as building blocks to the optimal solution of larger subproblems. While shared subproblems are a performance problem in top-down approaches, used together with optimal substructure, they are used to increase performance in *DP*.

3.2.1 Example

As a short illustration, we will use an example from the upcoming textbook [6]. Consider the problem of finding a *longest increasing sub-sequence (LIS)* in a (finite) sequence of integers, i.e. given a sequence $s =< a_1, \ldots, a_n >$, find a sub-sequence $< a_{i_1}, \ldots, a_{i_m} >$ s.t. $a_{i_j} < a_{i_k}$ and $i_j < i_k$ for all $j < k$. Clearly, for a sequence of length one, the longest increasing sub-sequence has length 1. For any sequence $< a_1, \ldots, a_n >$ of length $n > 1$ it is the maximum of

- the length of the LIS of $< a_1, \ldots, a_{n-1} >$
- 1 plus the length of the LIS of $< a_1, \ldots, a_{n-1} >$ that ends in an element smaller than a_n.

The first case occurs if $< a_1, dot, a_{n-1} >$ contains a LIS of $< a_1, \ldots, a_n >$, the second case occurs if a LIS of $< a_1, \ldots, a_n >$ ends with a_n. This yields the following, simple DP-algorithm for determining the length of any LIS in a sequence of integers:

- let $l_1 := 1$
- for i in $2, \ldots, n$ let $l_i := 1 + max_{j<i,\ a_j<a_i}\ l_j$
- return $max_{1 \leq j \leq n}\ l_j$

In this algorithm, the left-to-right order of execution makes it possible to compute l_i only once for every i although the value is used several times during computation. If, in the second step, for each i we save a reference to the index j that was used to compute l_i, we can also compute the actual LIS after determining its length. This is done by following the references starting from a sequence element which is the end of a LIS.

This example exhibits both characteristics we discussed earlier. It has shared subproblems because the LIS ending in a some element of a prefix of the given sequence is used to compute all the LIS of longer prefixes of the sequence. Furthermore, it has the optimal substructure property. If $< a_{i_1}, \ldots, a_{i_m} >$ is a LIS of $s =< a_1, \ldots, a_n >$, $< a_{i_1}, \ldots, a_{i_{m-1}} >$ is a LIS of elements of s less than or equal to $a_{i_{m-1}}$. We used these properties to in an efficient DP-algorithm for efficiently solving the problem.

While *DP* is usually used to compute an optimal result for an optimization problem, it can also be used as an improvement heuristic ([14], [15], [16]). This thesis uses *Dynamic Programming* as an improvement heuristic guaranteed to converge to an optimal result.

3.3 Graphs, Networks, Flow Problems and Residual Graphs

Graphs are an important tool in computer science theory. Many problems can be modeled as graph problems or network flows, which is why the basic algorithms covering these topics can be found in introductory algorithms textbooks like [5] and [6] or basic graph theory textbooks like [7] (which is also available in German [8]). This thesis contains the concepts of *residual graphs* and *augmenting paths*, which come from the *Ford Fulkerson* maximum flow algorithm. [5] has an excellent introduction to *Ford Fulkerson* in section 26.2. We will also use the concept of *Minimum Cost Flows*, which is covered in [10].

4 Formal Problem Definition

maximize

$$\sum_{0 \le t < T} \sum_{1 \le j \le m} v_{t,j}(x_{t,j}) \tag{1}$$

s.t.

$$x_{t,j} \in \mathbb{N} \qquad \forall t = 0, \ldots, T \; \forall j = 1, \ldots, m \tag{2}$$

$$\sum_{1 \le t \le T} x_{t,j} \le s_j \qquad \forall j = 1, \ldots, m \tag{3}$$

$$\sum_{1 \le j \le m} x_{t,j} \le fmax_t \qquad \forall t = 0, \ldots, T-1 \tag{4}$$

$$x_{t,j} \le fmax_{t,j} \qquad \forall t = 0, \ldots, T \; \forall j = 1, \ldots, m \tag{5}$$

where $v(t, j, x) \ge 0$,
$0 \le fmax_{t,j} \le fmax_t \;\; \forall t = 0, \ldots, T, \forall j = 1, \ldots, m$
and $v'_{t,j}$ monotonic decreasing

We are trying to maximize the profit from allocating (or *selling*) discrete goods from m different *sources* (also called *stores*), each of with a capacity $s_j > 0, 1 \le j \le m$, to T *locations* (or *time slots*). For each time slot t and source j we have a *decision variable* $x_{t,j}, 0 \le t < T, 1 \le j \le m$ that describes how many units from source j are allocated to time slot t. Each decision variable $x_{t,j}$ contributes to the objective function via a pricing function $v_{t,j}(x_{t,j})$ which we require to be non-negative. Furthermore, the first derivatives of these functions are required to be monotonic decreasing.

The values of the decision variables are subject to the following restrictions:

- Each store has a capacity (s_j for the j^{th}), which limits the number of items we can allocate from the resource stored in it (see equation 3).
- equation 5 states that the maximum number of items allocated at each time slot t is limited by a value $fmax_t$.
- An allocation can be constrained even further to a maximum number of items from a certain source j at a given time slot t by $fmax_{t,j}$ (equation 4).

The rationale for requiring the first derivatives of the pricing functions to be monotonic decreasing is the following: If one wants to sell one item of a good in a given time frame and has several offers, one will sell it for the best profit that can be realized. The next item will be sold for the second best price etc. Furthermore, it enables us to prove the correctness of the algorithm discussed in this thesis via an corresponding network flow problem.

Note that these pricing functions would allow different stores to contain different goods (although only one good per store). Constraint 5, however, suggests that those goods are somehow comparable, since it is a constraint across all stores.

Since we are dealing with multiple allocations, concave pricing functions and cross constraints, we call the problem *Cross Constrained Concave Multiple Allocation Problem* (*CCCMAP*).

Often the pricing can be expressed more intuitively by the profit realized from the k^{th} item sold from store j at time t, which we will call $\delta_{t,j}(k)$, defined for all $k \in 1 \dots min(fmax_t, fmax_{t,j})$. It is defined based on $v_{t,j}$:

$$\delta_{t,j}(k) := v_{t,j}(k) - v_{t,j}(k-1) \; \forall k = 1, \dots, \leq fmax_{t,j}$$
$$\forall t = 0, \dots T-1, \; j = 1, \dots, m, \; k \geq 1$$

or, vice versa,

$$v_{t,j}(k) = \sum_{1 \leq i \leq k} \delta_{t,j}(k)$$
$$\forall t = 0, \dots T-1, \; j = 1, \dots, m, \; k \geq 1$$

A feasible solution for this problem is called a *policy*, and consists of values for each of the decision variables $x_{t,j}$ for time slots $0, \dots, T-1$. Intuitively, it determines for each time slot how many units from which store to sell.

4.1 Policy versus States

There is another way to describe such a policy: For each time slot $0, \dots, T$ we specify remaining stock in each store, which we will call $r_{t,j}$ for each $t \in 0 \dots T, 1 \leq j \leq m$. The stock at time 0 is known from the problem instance, so it is redundant to specify it. However, it is much more human-readable to do so, which is why we will specify $r_{t,j}$ for $t \in 0, \dots, T$. We will call this the *state representation*.

Table 1 shows a simple example with 2 dimensions and 3 decision points that illustrates this correspondence. Since units can only be sold in our problem

	$x_{1,t}$	$x_{2,t}$	$r_{1,t}$	$r_{2,t}$
$t = 0$	1	0	3	3
$t = 1$	2	1	2	3
$t = 2$	0	2	0	2
$t = 3$	undef.	undef.	0	0

Table 1: Simple two-dimensional example

(due to the non-negativity of the decision variables), the stock is monotonic decreasing w.r.t. t. So opposed to the actual allocation decisions, where the only relationship between the values corresponding to a store over time are given by the capacity constraints, a store's storage is monotonic decreasing and it can be meaningfully plotted over time. Referring to this possible visual representation, we will call the development of stock over time a *trajectory*. Each legal policy corresponds to exactly one trajectory, and we will call a trajectory *legal* if it is the result of a legal policy. Formally,

$$r_{(t+1),j} - r_{t,j} = x_{t,j} \qquad \forall t = 1, \dots, T \;\; \forall j = 1, \dots, m \qquad (6)$$
$$r_{0,j} = s_j \qquad \forall j = 1, \dots, m \qquad (7)$$

For any given point in time, the vector containing the stock level of each store determines the state of the storage system at this time. We call such a vector a *store configuration* or just configuration.

We call the set of all possible configurations at a given time step, that is

$$R := \{0, \ldots, s_1\} \times \cdots \times \{0, \ldots, s_m\}$$

the *state space*.

4.2 A Corresponding Integer Linear Programming Formulation

There is another effect of the monotonic decreasing profits. Suppose we split up each integral decision variable $x_{t,j}$ into a vector of binary variables $y_{t,j,k}$, i.e.

$$x_{t,j} = \sum_{0 \leq k \leq fmax_{t,j}} y_{t,j,k}, \quad y_{t,j,k} \in 0,1 \quad \forall t, j, k$$

and consider the new optimization problem that is created when we replace the integral decision variables with the sum of their binary counterparts i.e.:

maximize

$$\sum_{0 \leq t < T} \sum_{1 \leq j \leq m} \sum_{0 \leq k \leq fmax_{t,j}} \delta_{t,j}(k) \cdot y_{t,j,k}$$

s.t.

$$\begin{aligned}
\sum_{1 \leq t \leq T} \sum_{0 \leq k \leq fmax_{tj}} y_{t,j,k} &\leq s_j && \forall j = 1, \ldots, m \\
\sum_{1 \leq j \leq m} \sum_{0 \leq k \leq fmax_{tj}} y_{t,j,k} &\leq fmax_t && \forall t = 0, \ldots, T-1 \\
y_{t,j,k} &\in \{0,1\} && \forall t, j, k
\end{aligned}$$

We were able to drop the constraint expressed in equation 5, because the number of binary decision variables implicitly enforces the maximum allocation value specific for both time and store.

Note that this is now an *integer linear optimization problem (ILP*. Each binary decision variable represents one allocation unit and contributes to the objective function with the profit corresponding to this allocation.

Now suppose we take an optimal solution to this problem and use it to generate a solution to our original problem by setting each integral decision variable to the sum of its corresponding binary decision variables. What could we say about this solution?

Obviously, it will be a legal solution, since the capacity constraints, the cross constraints and the individual constraints are enforced in the ILP-formulation.

Now let us look at the objective values of the ILP-solution and the solution to the original problem created from it. Since the binary decision variables corresponding to one integral decision variable are subject to the same cross- and capacity constraints, they can be picked independently from each other. This means that, if k decision variables corresponding to $x_{t,j}$ are of value 1 in an optimal LP-solution, they will be those with the highest profits.

It can easily be seen that if any one of these decision variables having value 1 has less objective value contribution than another one corresponding to $x_{t,j}$

which is set to 0, switching between those two allocations gives a higher profit without violating any constraints.

Formally, if the binary decision variables $y_{t,j,1}, \ldots, x_{t,j,k}$ corresponding to $x_{t,j}$ are set to 1 in the optimal solution, and their objective value contributions are

$$\{\delta_{t,j}(1), \ldots, \delta_{t,j}(k)\} \text{ where } \delta_{t,j}(a) > \delta_{t,j}(b) \Leftrightarrow a > b$$

then all x_{t,j_i} with objective value contributions at least v_{i-1} are set to 1.

However, since multiple binary decision variables can have the same objective value contribution, there may be several optimal solutions, having different (but the same number of) binary decision variables set to 1 (all with objective value contribution v_i).

When it comes to mapping an optimal ILP-result back to the original problem, though, this is not an issue. The integral decision variable $x_{t,j}$ is determined by

$$\sum_{0 \leq k \leq fmax_{t,j}} y_{t,j,k}$$

i.e. their sum, and the objective value contribution of $x_{t,j}$ is equal to the sum of the corresponding binary decision variables, because of the monotonic decreasing profits.

For the purpose of refutation, assume that there actually is a solution to the original problem with a better objective value than the optimal solution to the problem created by binary decomposition. For each integral decision variable $x_{t,j}$, order the corresponding binary decision variables by profit in non-increasing order. Let the resulting sequence be

$$< y_{t,j,o_1}, \ldots, y_{t,j,o_l} >$$

Let $x^*_{t,j} = k$ then let

$$y_{t,j,o_1} = \cdots = y_{t,j,o_k} = 1, \quad y_{t,j,o_{k+1}} = \cdots = y_{t,j,o_l} = 0$$

for all $0 \leq t < T, 0 \leq j < m$. Because of the non-increasing profits of the pricing functions, setting the k most valuable binary decision variables must yield the same profit as the integral decision variable in the original problem. Since this assignment of values creates legal solutions with the same objective value, we have refuted our assumption, and thereby proven that the ILP-formulation indeed yields optimal results for our problem.

5 A Simple Dynamic Programming Approach

To better understand the advantages of *Optimal Policy Iteration* we first look at a simple approach to solving *CCCMAP* using *Dynamic Programming (DP)* This chapter looks at the structure of the problem and explains why *DP* is a reasonable way of solving it. It describes this approach and analyzes the worst case running time.

5.1 Description

To understand why the problem is a reasonable candidate for *Dynamic Programming*, let us look at the state space. Given that at a given point in time each of our stores contains a certain number of items, the question is: how can we compute the value of this configuration?

- We can allocate, that is sell, some (or no) units from our stores at this point in time within our constraints
- ... and we can sell what is left in the stores later.

We have pricing functions for determining what profit we will realize by selling items at this point in time. Once this decision has been made, one has to quantify what "selling later" means in terms of profit. We can define it recursively as the value of the resulting configuration (i.e. the remaining stock) at the next time step.

This means that the value of a configuration is given as the optimal combination of

- legal values of this time slot's decision variables and
- the value of the resulting configuration at the next time step.

Since our problem has only a finite number of time steps, there is a last one, for which we have to make a decision as of how we estimate the value of each configuration. For the sake of simplicity, the formulation of *CCCMAP* assumes a value of 0 for all configurations at the end of the planning period.

Note that the recursion has shared subproblems. For example, in a one dimensional instance, having 3 units at time t could have been a result of having 4 units at time $t-1$ and selling 1 of them, or having 5 units and selling 2. The recursive nature and the shared subproblems make it an ideal candidate for *Dynamic Programming*.

With this knowledge, we have what we need for a DP algorithm. We will use a backward pass to determine the values of each configuration and save how the value was derived. In a forward pass we construct a policy to realize the optimal profit using this knowledge.

To do this, we define the value $v'_t(r)$ to be the value of a state $r_= < r_1, \ldots, r_m >$ from our state space $R = \{0, \ldots, s_1\} \times \cdots \times \{0, \ldots, s_m\}$ at time t. At the end of our planning period, all configurations have value 0, i.e. $v'_T(r) = 0 \ \forall r \in R$. Furthermore, we define $next_t(r)$ as the configuration at time $t+1$ following in an optimal trajectory starting at r at time t.

We use the backwards pass to compute these values.

1. Let $t := T - 1$

2. For each configuration $r \in R$ at time t we consider all configurations $r' \in R$ at time $t+1$. We already know the value of r' at time $t+1$, which is $v'_{t+1}(r')$. If $r_j - r'_j \leq fmax_{t,j}$ and

$$\left(\sum_{1 \leq j \leq m} (r_j - r'_j) \right) \leq fmax_t$$

the transition from state r to r' does not violate any constraints. Then the resulting value is

$$v'_{t+1}(r') + \sum_{1 \leq j \leq m} v_{t,j}(r_j - r'_j)$$

We define $v'_t(r)$ to be the best of these values. Furthermore, we remember which state at time $t+1$ we used to achieve this value, by setting the value of $next_t(r) := r'$.

3. If $t > 1$ then $t := t - 1$ and go to step 2

At the end of the backward pass, we have the values of each configuration at time 0, including the value of the configuration representing the capacities of our stores, i.e. $< s_1, \ldots, s_m >$. We can then reconstruct the optimal policy in a forward pass:

1. Let $t := 0, r =< s_1, \ldots, s_m >$
2. $r' := next_t(r)$
3. $x_{t,j} = r_j - r'_j \ \forall j = 1, \ldots m$
4. If $t < T - 1$ then $t := t + 1$ and continue at step 2.

5.2 Analysis

This algorithm is simple but inefficient. In each of the T time steps the algorithms visits all possible configurations, of which there are $\prod_{1 \leq i \leq m} s_i$.

For each of these configurations it checks all possible combinations of decision variables for the current time step. There are $O(\prod_{1 \leq i \leq m} s_i)$ such combinations. For each of these combinations it checks whether it is legal w.r.t. the constraints, and if it is, it computes a value, which needs time bounded by $O(m)$.

Finding the best value at time 0 takes $O\left(\prod_{1 \leq i \leq m} s_i\right)$ and following the next-Pointers takes $O(T)$ time. This yields a total time complexity of

$$O(T \cdot m \cdot (\prod_{1 \leq i \leq m} s_i)^2) \tag{8}$$

This expression is exponential in the number of dimensions (even though not explicitly), which is an effect of the well-known *Curse of Dimensionality*, and was to be expected.

The capacities of the stores appear in a product, which is taken to the second power. This is very undesirable: It would be intuitive for the capacities to be a linear factor in the running time. Here the effect of increasing the capacity of stores is significantly bigger.

For example, consider an instance I with m stores and x units per store, and an instance I' of the same dimensionality but $2x$ units per store. According to 8, the worst case running time of I is

$$O(T \cdot m \cdot x^{2m})$$

as opposed to

$$O(T \cdot m \cdot (2x)^{2m}) = O(\cdot T \cdot m \cdot 2^{2m} \cdot x^{2m})$$

for I', which is bigger by a factor of 2^{2m} or 4^m. More general, this means that multiplying all capacities of an instance with m dimensions by a factor of k increases the worst case running time of the simple DP algorithm by a factor of k^{2m}. Because of this dramatic change, running instances that are large w.r.t. capacities becomes infeasible, even if their number of dimensions is small.

It would therefore be desirable to find a method for solving these problems that, in its running time, separates the two components discussed — the number of stores their capacities. We still expect the number of stores to be an exponential factor in the running time, however, we would like the capacities to contribute just linearly.

6 *Optimal Policy Iteration*, a More Efficient Algorithm

We have considered a simple *Dynamic Programming*-approach to solve *CCCMAP* and discussed why its running time is unreasonably big. This chapter introduces *Optimal Policy Iteration*, an algorithm found by Alfred Kalliauer, which reduces the impact of capacities on running time. We will look at a detailed description of the algorithm, as well as an analysis of worst case running time.

6.1 Description

In the definition of *CCCMAP* we required the first derivative of the profit functions to be monotonic decreasing in every dimension, i.e. given a dimension and a time step, the $n+1^{st}$ allocation may not yield more profit than the n^{th}, or formally

$$\delta_{t,j}(n) \geq \delta_{t,j}(n+1) \quad \forall t = 0, \ldots, T-1, \forall j = 1, \ldots, m, \forall n = 1, \ldots, fmax_{t,j} - 1$$

This closely follows the metaphor of a market: When one has several units of some good, and a number of buyers offer different prices, one will first sell as many units as possible for the best price, then the second best price etc.

On the other hand, if we do not require this property of the pricing functions, small changes in the capacity of a store can have a big effect on the optimal solution. Let us consider table 6.1 that shows an instance of a problem that is identical except for not having this constraint and see what can happen if we increase the capacity of one dimension by one unit.

x	$\delta_{01}(x)$	$\delta_{11}(x)$
1	1	10
2	1	10
3	1	10
4	100	10

Table 2: Pricing function possible without restrictions

Clearly, for up to 3 units the optimal solution would be to allocate all units to time step 1. However, when there are 4 units, the optimal solution allocates all units to time step 0. It is easy to see that this scheme can be used to build instances where adding one unit to the capacity of a store creates arbitrarily big changes in the optimal solution by assigning an excessively big profit to a single allocation which can only be realized after doing a number of unprofitable allocations.

Requiring the profits to be non-increasing changes this. As an illustration, consider the simple *CCCMAP*-instance of the same size shown in table 6.1

In this instance, the first three available units are best allocated time step 1. Again, the fourth allocation at time step 1 is worth less than the fourth allocation at time step 0. However, since the profits are non-increasing, the same holds for the first three allocations at time 0. Hence, when a fourth unit

x	$\delta_{01}(x)$	$\delta_{01}(x)$
1	10	11
2	10	11
3	10	11
4	10	1

Table 3: Example with restriction on pricing function

becomes available, it is allocated at time step 0, yielding an optimal solution that differs only slightly from the one with three units.

We will prove later that this similarity of the optimal solutions for similar problem instances is indeed an effect of the non-increasing profits. Assume we have an optimal solution for a *CCCMAP*-instance I. Consider the instance I' that is the result of adding one unit to the capacity of one dimension and compare the optimal solutions. We can prove the following statements:

None of the values of the decision variables in the optimal solution to I differs by more than one unit at any time from those in I'. Or, more formally, let $x^*_{t,j}$ denote the value of the decision variable at time t and time j of an optimal solution for I and $x^{*'}$ i.e.

$$|x^*_{t,j} - x^{*'}_{t,j}| \leq 1 \quad \forall t = 0, \ldots, T-1, \forall j = 1, \ldots, m$$

This means that in the decision space, there is an neighborhood property between the optimal solutions of similar problem instances.

But more important, the trajectories of the two optimal solutions do not differ by more than one unit at any time and dimension, i.e.

$$|r^*_{t,j} - r^{*'}_{t,j}| \leq 1 \quad \forall t = 0, \ldots, T-1, \forall j = 1, \ldots, m$$

This means that there is even a neighborhood property in the state space of the optimal solutions. Figure 1 illustrates this for an example with one store.

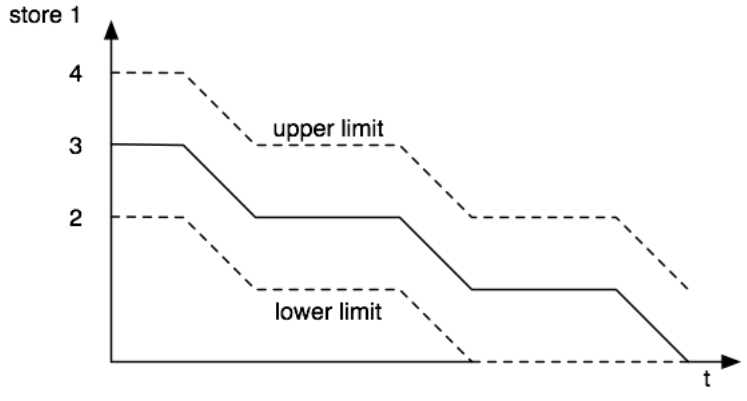

Figure 1: Limited search space around an optimal solution.

Given an optimal solution for three units, the optimal solution for four units

cannot be further than one unit away at any time step. Obviously, it cannot be negative either.

The simple DP-algorithm solved *CCCMAP* by searching the whole state space in the backward pass and constructing the optimal trajectory in the forward pass.

Given such a strict neighborhood property in the state space, we can find the optimal trajectory of the slightly bigger instance via a *Dynamic Programming*-search within a small neighborhood of the old optimal solution by restricting the starting- and end-points of the backward pass to the configurations that do not differ from those of the old optimal solutions by more than one unit per dimension.

To actually perform this search, we project the state space into another space of the same dimensionality, i.e. m dimensions for state and 1 dimension for time. The projection is chosen such that the old optimal solution has value 1 in each store at each time step in the new state space. When we consider store values of 0 to 2 in this new state space, 0 always means "one unit less than in the old solution", 1 means "the same" and 2 means "one unit more". This is illustrated in figure 2, where the such a projection. The optimal solution after adding one unit is indicated in magenta.

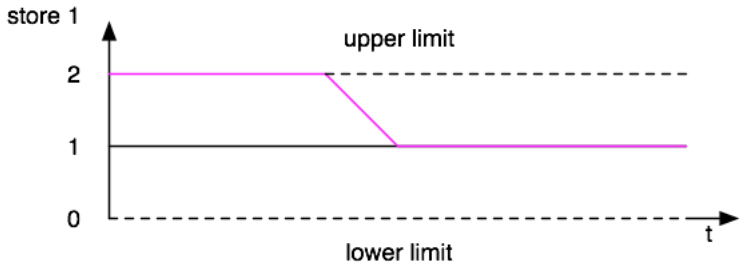

Figure 2: Finding an optimal solution for a larger sub-problem-instance.

We can now easily represent this search space in a multi-dimensional array, and perform a DP-search on it, allowing only state values of 0 to 2 in each dimension. In the backward pass we can check the constraints by projecting the starting- and end-point of each state transition we consider back to the original state space.

After finishing the backward pass, we perform the forward-pass to find the optimal solution to the new problem. Then we project this solution back into the original state space. For the example used before, figure 3 shows the new optimal solution projected back into the original state space.

The outlined procedure generates an optimal solution to one problem instance based on the optimal solution of a slightly smaller one. This poses the question of which solution to start with. Alfred Kalliauer devised a very elegant solution to this[ref]: For every *CCCMAP*-instance there is a trivial instance with the same number of time-steps and dimensions, but a capacity of zero units in all stores. The optimal solution to this problem is simple: Not allocating anything, i.e. all state- and decision variables are set to zero.

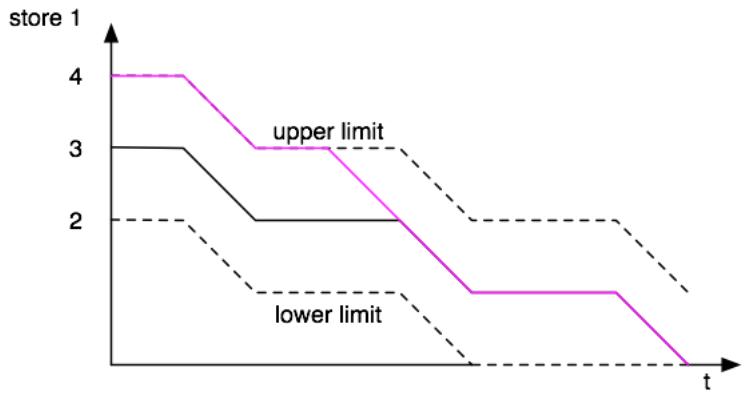

Figure 3: Projecting the new optimal solution to the actual state space.

From there we can start incrementing the capacity of the dimensions until we have reached the capacities of the original problem instances (or no capacity increase increases the objective function value. Since the optimal solution is reached via a series of optimal solutions to smaller subproblems, Alfred Kalliauer called this algorithm *Optimal Policy Iteration (OPI)*.

We will call one DDP search step around an optimal solution, yielding an optimal solution for a bigger instance, an *OPI-iteration* or just an *iteration*. When we look at the DDP search space for an iteration for an example with two dimensions, it looks very much like a tube (see figure 4). Following this visual metaphor, we will call the DDP search space (for problems of any dimensionality) the *search tube*.

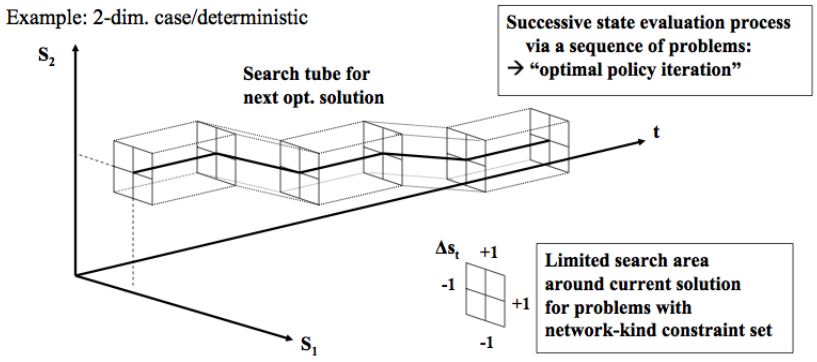

Figure 4: 2D Example (Source:[2])

6.2 Summary

The following description summarizes *OPI*:

1. For a given *OPI*-instance I define the trivial instance I_0 with all capacities set to 0.
2. Set the current optimal solution to 0 in all state variables at all time steps.
3. Increase the capacity of one store in I_i by one unit, yielding an instance I_{i+1}.
4. Project the old optimal solution into an $m+1$-dimensional array such that all state variables are set to one at all time steps
5. Perform a DP-backward-pass on this array. To check for constraint violations, compute the value of starting- and end-points of state transitions in the original state space.
6. Perform the a forward pass to find the projection of the optimal solution to I_{i+1}.
7. Project this solution back to the original state space to obtain the actual solution.
8. Continue with step 3 unless all capacity limits are reached or the capacity increase does not improve the objective function value.
9. Output the optimal solution to I.

6.3 Analysis

Starting from the trivial solution of I_0 we do at most $\sum_{1 \leq j \leq m} s_i$ local DP searches. Each of these searches has T steps in which all possible combinations of 3^m current configurations and the 3^m configurations of the next time step are evaluated. We, again, assume that evaluating one of these time combinations takes $O(m)$ time. This yields an overall time complexity of

$$\sum_{1 \leq j \leq m} s_i \cdot T \cdot m \cdot 3^{2m} \tag{9}$$

Let us compare this to the running time of the conventional DP-Algorithm, which is

$$O(T \cdot m \cdot (\prod_{1 \leq i \leq m} s_i)^2) \tag{10}$$

Two observations can be made concerning running time:

- The number of dimensions is still an exponential factor (3^2m).
- However, the capacities of the stores are now a linear factor, vs. being in the squared product.

6.4 A Simple Example

7 Proof of Correctness of OPI

Optimal Policy Iteration uses a neighborhood property of *CCCMAP* to iteratively build an optimal solution to a given instance. This chapter contains a thorough mathematical proof this property and shows the correctness of the algorithm.

7.1 Network Flow as an Abstraction

The inspiration for the abstract model that OPI is solving were hydro storage power plants, which work by harnessing the potential energy of water that is released when it flows between different elevations. Since a physical flow is present in such systems, network flows would be one way of thinking when trying to find an abstraction for such a system. Indeed, this is was Alfred Kalliauer's starting point was and finally enabled him to devise the OPI-Algorithm.

For a better understanding of the flow network involved in the proof of correctness for OPI, let us have a look at how hydro storage planning in general can be represented by a flow network, gradually going into more detail.

Figure 5 shows a very abstract view of a hydro storage system consisting of just one store in the context of a flow network (i.e. with a source and a sink). There is a source of water, and a target where the water flows after passing the power plant, which we will represent with a sink, as known from network flow theory. Here, the actual hydro storage system and the turbines that are generating the power are represented by rectangular nodes.

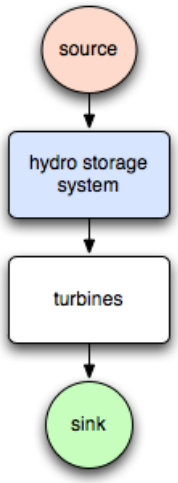

Figure 5: Abstract view of a hydro storage system.

For the sake of simplicity, let us stop considering the practical aspects of generating power for the time being, since it is not relevant for the planning, i.e. we remove the turbine from this system.

Since a standard flow network has no intrinsic representation of both time and storage, we choose to make this explicit using auxiliary nodes. We do this

by splitting up the storage system into several nodes, each one representing the store at one time step in our planning period (see figure 6 for a graphical representation of such a network).

Every node has two incoming and two outgoing arcs. The incoming arcs are the natural inflow of water, and the water that has been stored until this time step. The outgoing arcs represent the choices one is faced with such a reservoir: Use some of the water in the store for generating electricity (represented by the arcs leading to the sink) and/or keep water in the store (represented by the arc to the store at the next time step). Maximum capacity on those two arcs can be used to limit both the amount of water that can be used to generate electricity and the amount of water that can be stored at any time.

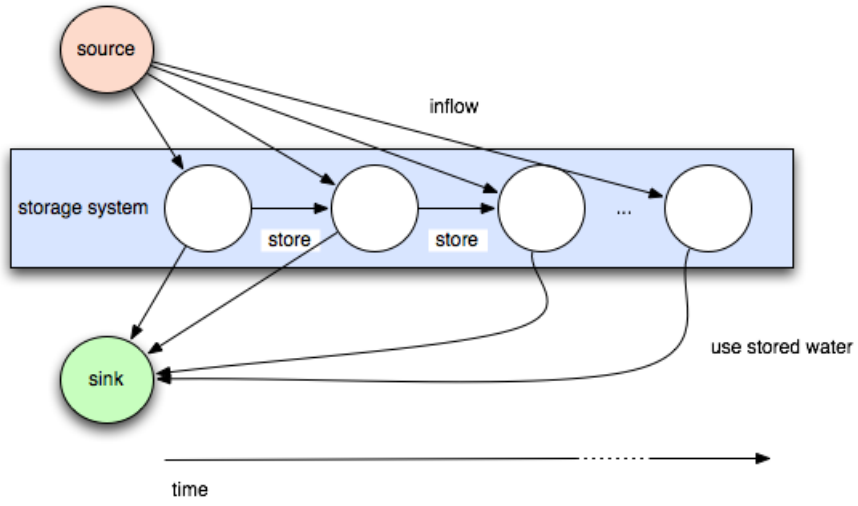

Figure 6: Representing water storage in a network

In our problem, we are not considering natural inflow, we just agree on an amount of water that is stored in the beginning and used in course of the planning period. This means that the amount of water in the store will only decrease. Since we suppose our problem instance is correct, meaning we do not start with more water in the store than can be stored, storage limits are not an issue of our planning. Hence we can use just one node for each store, with only one incoming arc that represents the water that is already in the store.

What we do want to consider, however are variations of the electricity markets. Hence, our network should be able to enforce a different maximum number of allocations at each time step, so we need to keep at least one outgoing arc for each time step. This is illustrated in figure 7.

This would be sufficient to enforce the constraints corresponding to a system with only one store. *CCCMAP*, however, can have several stores. We will have one store node for each of them. To enforce the constraints covering the total amount of electricity generated in all power plants of the system at a given time step (the *cross constraints*, we need the flows corresponding to this time step from all the stores to influence each other in some way. The solution for this is

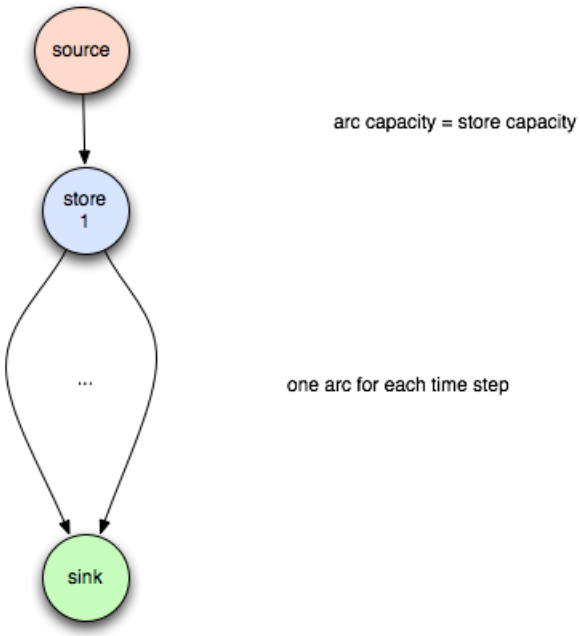

Figure 7: Representing several stores and cross constraints

adding one node for each time step, which we will call *time nodes*. They have incoming arcs from all stores. The flow between a store node and a time node is the amount of electricity generated at the corresponding store and time step, i.e. it represents one decision variable. Their outgoing arcs go to the sink, and their capacity is this time step's cross constraint, satisfying the requirement that the flows from all stores at this time step influence each other. Such a network is shown in figure 8.

This network can be used to enforce all the limits of the problem, namely

- store capacity (as incoming arcs to store nodes)
- cross constraints (as outgoing arcs of the time nodes)
- time/store specific limits (as arcs between time and store nodes)

However, to create a complete representation of our problem, we need some way to model the objective function.

7.2 Adding a Representation of the Objective function

To represent the objective function, we add costs to the network in figure 8 and add some more refinements. We chose to use a *minimum cost maximum (integral) flow* formulation. The *minimum cost maximum flow* problem is to find a maximum flow of minimal cost. This means that we now, as desired, have two different aspects to this problem, namely arc capacities and weights, which

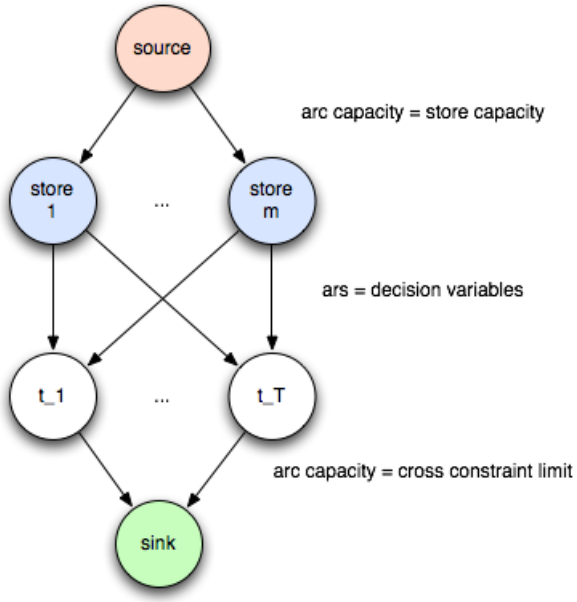

Figure 8: Representing several stores and cross constraints

can use those to model both constraints (via arc capacities) and the objective function (via weights).

7.3 Final Network Layout for the Proof

We define a function from instances $\mathcal{I}$ of *CCCMAP* to a subset of all weighted flow networks in a way that allows us to map each instance of *CCCMAP* to a flow problem, of which the optimal solution, a minimum cost maximum flow (which we will call F), can be mapped to the optimal policy P for our instance.

The overall layout of our network is illustrated in figures 9 and 10. The dashed lines indicate several arcs (that will be illustrated in detail in later figures). Arcs are labeled with two values, separated by a horizontal line: Their capacity and their weight. A label "3|2" means a capacity of three and a weight of 2. Notice the following properties:

- The network contains both a source s and a sink t.
- Each store is represented by one node (which we will call *store node*).
- Each time step is represented by one node (which we will call *time node*)
- Each decision variable is modeled by several arcs between store nodes and time nodes which are indicated by dotted arrows in figure 9. Each of these arcs represents one allocation decision
- The contribution of each allocation decision to the objective function is represented by assigning it negative costs. If the profit of an allocation is p the corresponding arc's weight is $-p$.

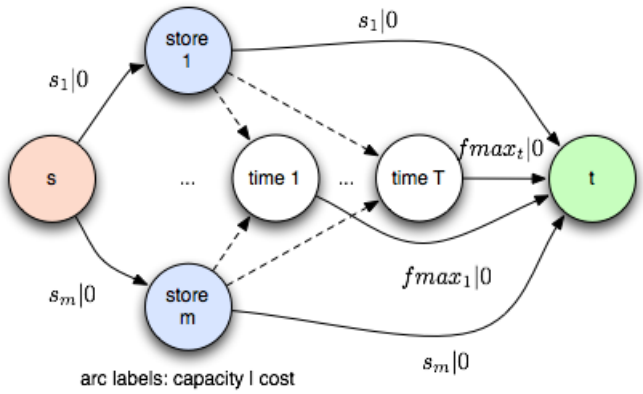

Figure 9: Final network (overview)

- The objective function is represented by the sum of these (negative) costs.
- There is an additional arc for each store, going from the corresponding store node to the sink, having the capacity of the store but no cost. Units that are not allocated contribute to the flow here.

7.3.1 Stores and Store Capacities

Each store corresponds to exactly one store node. In the drawings of the graph, the store nodes are colored blue and labeled with their numbers.

There is one arc between the source and each store node. This arc is used to model the capacity constraints. To achieve this, its arc capacity is identical to the capacity of the corresponding store. We are not using these arcs to model the objective function, so we set their weight to zero.

7.3.2 Time steps

Each time step is represented by one *time node*. Their incoming arcs are those corresponding to the decision variables (which are explained in more detail shortly). Their outgoing arcs go directly to the sink and their capacity is equal to the value of these time steps' cross constraints (thus enforcing them).

7.3.3 Objective Function

The objective function of our problem is computed as the sum of the pricing functions which assign a gain to the value of each decision variable. In the network each decision variable is represented as *several* arcs between a store and a time node as illustrated in figure 10, one arc for each unit that can legally be allocated from the corresponding store at this time step with a positive objective value contribution. That is, there are $fmax_{t,j}$ arcs from store node j to time node t. One arc corresponds to one allocation. In other words: Each arc corresponds to one binary decision variable in the ILP-formulation.

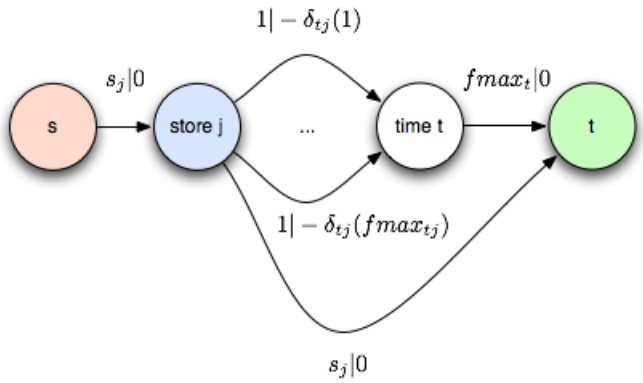

Figure 10: Representation of the Decision Variables and Objective Values

Each arc's capacity is obviously 1 since it corresponds to the allocation of 1 unit. The cost represents the profit realized from this allocation, i.e, for the n^{th} arc between store j ant time t it is

$$-\delta_{t,j}(n) = -(v_{t,j}(n) - v_{t,j}(n-1)).$$

Since in the *minimum cost maximum flow* problem we are looking for a maximum flow of *minimal* arc cost, and we want to *maximize* the objective value, we must multiply the objective value of an allocation by -1 to represent it as arc cost. This way, more value translates to less edge cost. Note that, at this point, we are only theoretically looking at the solutions of the flow problem, not at the algorithmic challenges involved in its solution that come from the negative arc costs.

Just like in the ILP-formulation, splitting the decision variables into several arcs requires the pricing function's profits to be non-increasing w.r.t. k. Since all the arcs between a store and a time-node are of the same capacity, a minimum cost maximum flow will contain those of lowest cost (that is, highest objective value). This is exactly what we require from our pricing functions: The first allocation of a good at the given time step is the most valuable etc. This way, the arcs are chosen in an order that ensures equal objective value contribution compared to the original problem.

So if there is a total flow of value f between store j and time t, the value of $x_{t,j}$ will be f. The objective value of the decision variable corresponding to a time and a store is then the sum of the weight of the corresponding arcs, multiplied by -1.

We have just established that the value of an allocation decision is represented correctly in the sum of costs of the flow problem. However, to actually establish equivalence of our flow problem with the allocation problem, we need to show one more thing: That requiring a maximum cardinality flow will not lead to a suboptimal allocation.

It is quite obvious that, if making space for a unit (by shifting other units to less profitable allocations) costs more than its allocation realizes as a profit, not allocating it at all is better for the overall objective function. This is why we

have to make sure that not allocating a unit at all contributes to the cardinality of the flow just as the unprofitable allocation would do.

This property is ensured by the arcs that directly connect the store-nodes with the sink, as shown in figure 10. These arcs have the full capacity of the store, but no cost. So any inflow of the store nodes can be sent directly to the sink without having an impact on cost or the decision variables. Since these special arcs will appear several times in the proof, we will give them a name: *free arcs*, as an allusion to their cost of 0. This makes allocations that reduce the objective value unnecessary for ensuring a maximum flow.

7.4 Summary: Equivalence to the Problem

The following list summarizes the arguments, why we indeed can produce a flow network N for every one of our problem instances I where the minimum cost maximum flow F maps to the optimal policy P:

- The integrality constraint is trivial since we require an integral flow.
- The constraint that we can only allocate as many items as we actually have is represented by the inflow capacity of the store nodes and skew symmetry of flows (see figure 9).
- The constraints about the maximum number of allocations at a given time is handled by the outflow capacity of each time-node and skew symmetry of flows (see figure 11).
- The constraints specific to both store and time step can be enforced by restricting the number of arcs between the corresponding nodes, each of which has capacity 1.
- The objective function is represented by the (negative) arc costs.
- Requiring a maximum flow has no negative impact on the objective values.

7.5 Necessary and Sufficient Criteria for a Minimum Cost Maximum Flow

When is a flow a *minimum cost maximum flow*? The necessary and sufficient criteria are:

- It is a maximum flow.
- There are no cycles of negative weight in the residual graphs. Otherwise, pushing flow along these cycles would reduce the cost.

7.6 Using the Equivalence to Prove Correctness of OPI

Optimal Policy Iteration works by using an optimal policy P_i for an instance I_i to calculate optimal policy P_{i+1} for I_{i+1}, where I_{i+1} differs from I_i only in the capacity of one store, which is larger by one unit.

Each such step is made by a local search around the trajectory corresponding to P_i, which covers all trajectories that differ from that of P_i in at most 1 unit per dimension in each of the time steps.

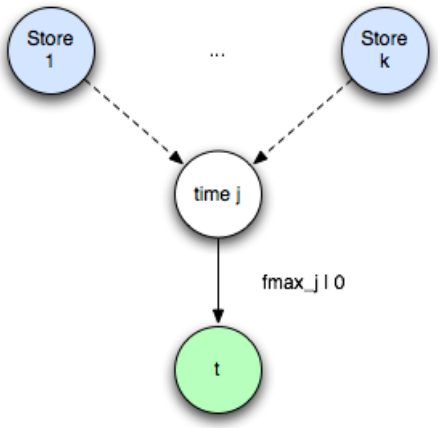

Figure 11: Flow per Time constraints

The starting point for OPI is the trivial problem I_0 where all the capacities are 0, and where the obvious optimal solution is not selling at all.

To prove the correctness of OPI, we need to show two things with this network equivalence:

- When doing an OPI-step, the new optimal solution is indeed within our search space.
- The algorithm terminates, i.e. finitely many OPI-steps are necessary for each *CCCMAP*-instance.

7.7 One OPI Iteration - Inductive Arguments

To do this, we will look at two flow problems corresponding to a *CCCMAP*-instance I, N_i and N_{i+1}, assuming that we have an optimal solution F_i to N_i and look how F_{i+1} differs from it. So this is an induction proof, with the trivial problem I_0, and its corresponding network N_0 as the basis, and the capacity increase as the induction step.

So how can the optimal flows differ after increasing the capacity of one store by one unit?

7.7.1 Augmenting Simple Path Always Exists

For the sake of a rigorous discussion, we must check whether it is possible that the maximum flow does not change at all. In our network this is not the case. Since we have the *free arcs* between the store-nodes and the sink that have the whole capacity of the corresponding store, we can always send flow directly along those. This increases the cardinality of the flow by one via a simple augmenting path. Hence, there always is an increase of the total flow.

Lemma 7.1. *Every store capacity increase creates a simple augmenting path in the corresponding flow network.*

7.7.2 Capacity of the Augmenting Path

Lemma 7.1 has established that the capacity increase always creates an augmenting simple path. Let F_i be a maximum flow for N_i. N_{i+1} differs from N_i only in one arc, for which the capacity is one unit bigger in N_{i+1}. Assume we apply F_i to N_{i+1} and there exists an augmenting path in the residual network with a capacity greater than one. Only one unit of that capacity can be due to the difference between N_i and N_{i+1}. This means that there must have been an augmenting path in the residual graph of F_i applied to N_i, although we assumed F_i to be a maximum flow for N_i. Hence, the maximum capacity of an augmenting path in the residual graph for F_i to N_i is one unit.

7.7.3 Choosing a Path

Since Lemma 7.1 tells us that there is always a simple augmenting path, we can ignore cases of augmenting paths that contain cycles and treat the cycles independently later in the proof. From all these *simple augmenting paths*, we chose the one with *minimal cost*, and if there are several of them, we choose one of *shortest length*.

7.7.4 No Changes

Note that if the minimal cost that we can achieve is 0, the shortest augmenting path is the one going from source to the store-node of, and then directly to the sink via the free arc. Mapped back to the *CCCMAP*-instances, this does not change the allocation decisions. In other words, when our criteria choose this path, having more unit from this store available has no impact on objective value at this point in the algorithm, i.e. this unit will not be allocated.

7.7.5 Allocations and Cascades

In many cases, though, the best augmenting simple path will look something like the path illustrated in figure 12 and have negative cost (i.e. positive objective value). The capacity increase in store i will lead to one more unit of i being allocated at some point x in time. This may, but need not, make it necessary to remove a unit from some other store, say store j, to be removed from time x, because of cross constraints. This unit may then be allocated at another time and so on. We will call such a chain reaction represented by a simple path a *cascade*.

These cascades will at most move one unit from each store. If a unit from a store j is moved, and later in the cascade another unit from j is moved, the situation would look as shown in figure 13. Since we defined them as the effects of simple augmenting paths, this will not happen in a cascade (we will explicitly treat cycles later, though).

A cascade is therefore finite, and there are two ways for it to end.

- The last unit in the cascade is actually allocated. This will look like in figure 12.
- The last unit in the cascade is *pushed out* of the allocation. Figure 14 is represented by the flow being pushed through the free arc of this unit, directly into the sink.

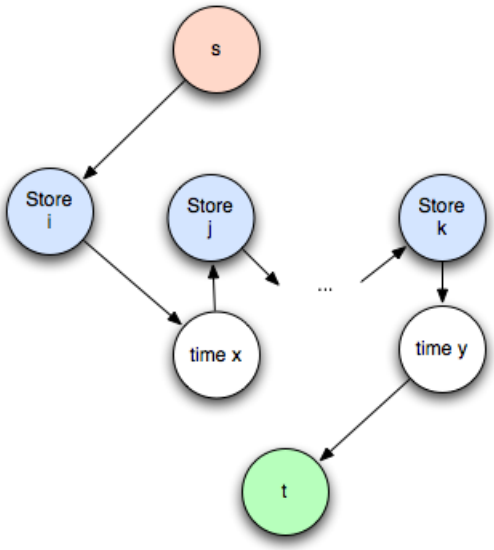

Figure 12: An Augmenting Path

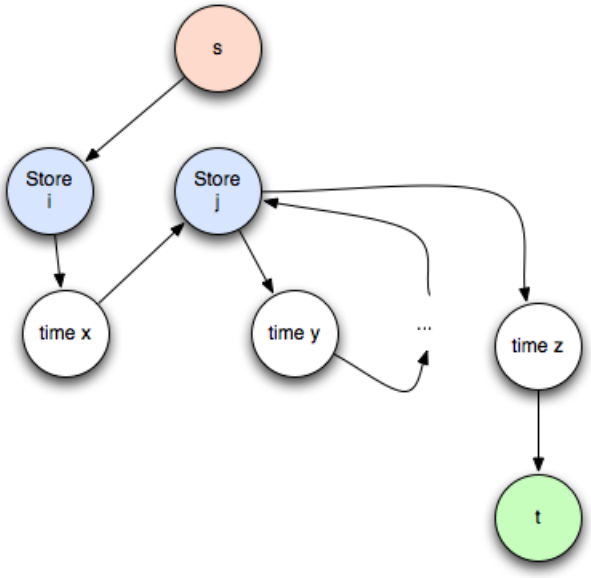

Figure 13: Moving more than one unit from a store yields a cycle.

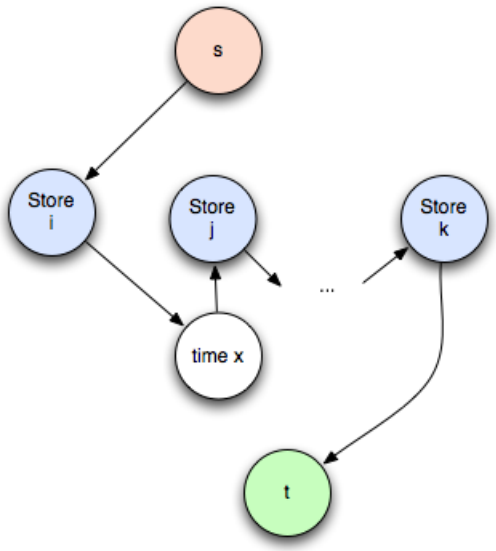

Figure 14: An Augmenting Path pushing a unit out of the allocation

7.7.6 Maximality after Pushing a Flow

Along this shortest simple augmenting path of minimal value, we push a flow of value 1. Is the resulting flow maximal and of minimum cost? We will look at these two aspects separately.

Maximal cardinality is easier to prove. Assume, for contradiction, that there is one more augmenting path in the graph, because of the layout, it must go from the source, via one store node, and then any other number of nodes, to the sink. This means that at least one arc between the source and a store node is not yet saturated, independent of the capacity increase in this step, i.e. F_i applied to N_i did not saturate this arc as well. Together with the corresponding free arc, this creates a path augmenting F_i over N_i, even though we assumed it to be maximal. Together with Lemma 7.1 this yields.

Lemma 7.2. *Every capacity increase creates at least one augmenting path in the residual graph of the flow network. The capacity of any such augmenting path is exactly one unit. After pushing a flow of one unit along any one of these paths, the resulting flow is a maximum flow.*

7.7.7 Minimum Cost after pushing a Flow

Having established that we now indeed have a maximum flow, we need to check whether it is of minimal cost. The only way to reduce cost now would be a negative cost cycle.

If, at this point, there exists a negative cost cycle along which flow can be pushed, and which is independent of the augmenting path that we just discussed (i.e. it existed before the flow was pushed along the path), it would have also

existed without the capacity increase. This contradicts the assumption of F_i being the optimal solution for N_I.

When exactly does such a cycle exist independently of the changes made by pushing flow along the augmenting path? It is easy to see that this is the case when it shares no vertices or arcs with the augmenting path. Even having vertices in common with the augmenting path is no problem by itself. When the augmenting path allocates a unit from store j to time t, it takes the one of the most profitable arc between the two vertices, so it is not more profitable to allocate there after flow was pushed along the path.

This means that only a cycle that visits two vertices in the opposite direction of the cascade may have been made possible by it: If, before pushing flow along the cascade, k units of j were allocated to t, and afterwards only $k-1$ are, this potentially more valuable (and now possible) allocation creates the possibility of having a negative cost cycle where once there was not. This is illustrated in figure 15.

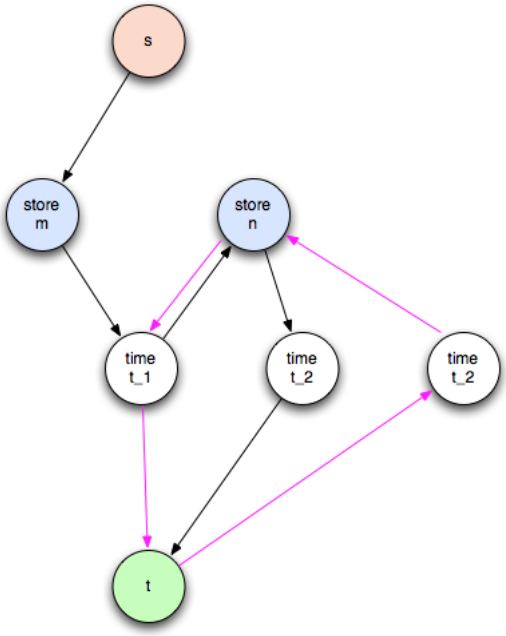

Figure 15: A negative cost cycle enabled by an Augmenting Path

As we see in figure 16, the fact that the allocations run in opposite direction, actually splits them up in a path and a cycle that are independent of each other. However, it is not immediately obvious how to make assertions about the costs concerning these two new structures. What we do know is that, by assumption, the initial augmenting path from figure 15 had minimal cost c_1 (a negative number). Furthermore, it allowed for a negative cost cycle with cost c_2. Let us denote the total cost of these two changes by $c = c_1 + c_2$. Note that, since the cost of the common edge once contributes to the cost both positively (de-allocation by the augmenting path) and negatively (allocation by the cycle), it

is of no concern to the sum of the total cost of the changes.

Taking another view of the same changes, i.e. seeing an augmenting path and a cycle independent of each other, obviously does not alter the cost of the changes which remains c. Let us denote the cost of the augmenting path from figure 16 with c'_1 and the cost of the cycle as c'_2. Obviously, $c = c'_1 + c'_2$.

This leaves two possibilities for the values of c'_1 and c'_2. One is that the cycle has negative cost, which means that there is a negative cost cycle independent of the capacity increase, which contradicts the optimality of F_i for N_i. The other is that the cycle has non-negative cost, i.e. $c'_2 > c_2$. Since $c'_1 + c'_2 = c_1 + c_2$, this yields $c'_1 < c_1$, which contradicts the assumption that the augmenting path was the one with the lowest cost.

By contradiction, we proved that after pushing a flow along the shortest augmenting simple path of minimal cost transforms one minimum cost maximum flow into the next. This yields

Lemma 7.3. *When applying F_i to N_{i+1} there always is an augmenting simple path, and by choosing the shortest augmenting simple path of minimal cost and pushing a flow of one unit along this path, we get F_{i+1}. F_{i+1} on N_{i+1} has no augmenting paths or negative cost cycles, it can be mapped on the optimal solution to I_{i+1}.*

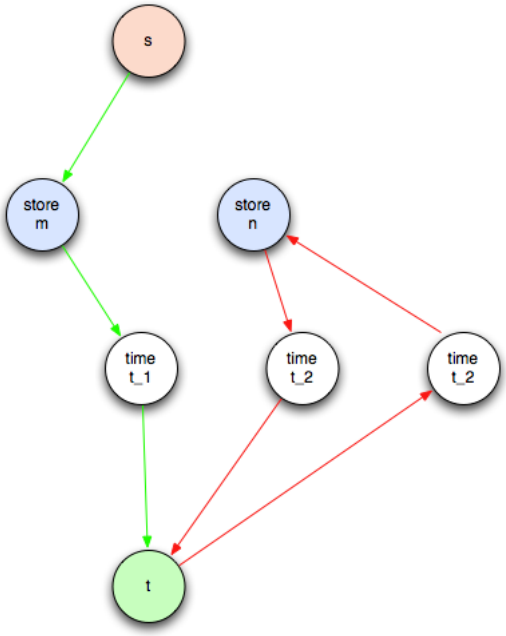

Figure 16: Result: one path, one cycle

7.8 Locality

Theorem 7.4. *Given an optimal policy P_i for CCCMAP-instance I_i:*
When adding one unit to the capacity of one store, yielding instance I_{i+1}, in

state space its optimal solution P_{i+1} differs from P_i by at most one unit per dimension at each time step.

For the new trajectory to differ from the old one by more than one unit in any dimension, it is necessary to allocate two or more unit at one dimension more or less than on the previous policy. This must yield an objective value not achievable within the described bounds.

What this means in terms of the network flow is that one store has to have two more units of inflow or outflow (which are equal requirements because of skew symmetry). However, since we know that a simple augmenting path can make the transition between two optimal solutions, this requirement would imply the existence of a cycle (see figure 17, which we have proven to be impossible.

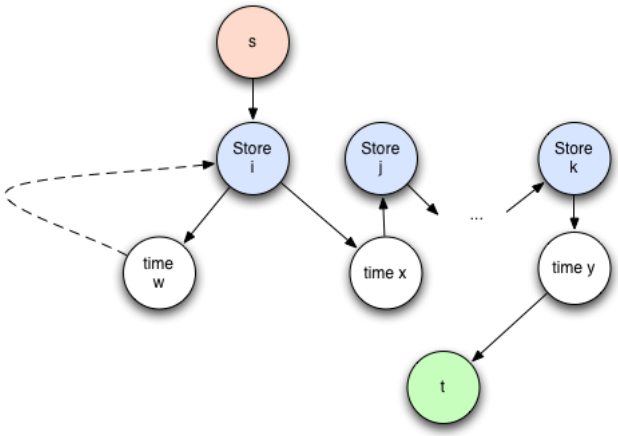

Figure 17: Two incoming edges imply the existence of a negative cost cycle

Corollary 7.5. *The local DP search space of OPI is sufficient.*

7.9 Termination

Theorem 7.6. *After a unit from store j is either pushed out of an allocation when increasing another store's capacity, or increasing j's capacity causes no change in allocation in any OPI-iteration, increasing the capacity of store j in any later OPI-iteration will cause no changes in the optimal policy.*

If a unit from store j is pushed out of the allocation, or a capacity increase of j does not change the optimal solution, the free arc of j has a flow value of at least 1 unit. Hence, if allocating a unit of j would become profitable later, this would result in a negative cost cycle, which we have proven to be impossible.

Corollary 7.7. *After at most $\sum_{1 \leq j \leq m} s_j$ iterations, OPI will terminate with an optimal solution to a given CCCMAP-instance.*

8 Order of OPI Iterations

As described and proven earlier, each OPI Iteration serves to calculate the optimal solution to a problem, given the optimal solution of a problem that differs only by one unit in the capacity of one store. If we write the store capacities of the problem P one is trying to solve in vector form, that is $< s_1, \ldots, s_m >$, the OPI process starts with a problem P_0 with capacities $< 0, \ldots, 0 >$.

Increasing the capacity of one store corresponds to adding one of the vectors $\{e_i; 1 \leq i \leq m\}$ of the natural basis of $\mathbb{R}^m$. In $\mathbb{R}^3$, e_2 is $< 0, 1, 0 >$. Hence, an OPI iteration finds the optimal solution of the problem $P_i + 1 = P_i + e_k$ given the optimal solution for P_i for some i and $1 \leq k \leq m$.

To be more precise, the backward pass of an OPI iteration, given the optimal solution to P_i, calculates the information needed to calculate the optimal solutions to $P_i + e_k$ for *all* $1 \leq k \leq m$ (actually also for all $P_i - e_k$, but this is not useful), and even the objective value associated with each of them. The forward pass then calculates the actual trajectory. This brings up an obvious question: Which of the neighboring problems shall be chosen for the forward pass, and thereby for the following iterations?

To answer this, let us first look at when *OPI* actually stops. The *OPI*-process is terminated when in each store

- the capacity constraint is met for each store.
- no matter in which store we increase the capacity, the optimal solution does not change, i.e. the optimal solution can be obtained with less units than available.

In the previous section we have shown that after an increase of one unit from a store is not profitable anymore, or one unit of a store is pushed out of the allocation, we do not need to consider adding a unit from this store later. There is no "oscillation" between different solutions.

Hence, we can choose freely in which order we want to add units. Two strategies come to mind quickly.

- The first strategy is one dimension after the other (which we will call *by dimension* for brevity), i.e. one increases the capacity of the first dimension until it yields no more profit, and goes on with the second, etc.
- The first strategy is *best increment*, where we use our knowledge of the objective values of all the neighboring problems to choose the one with the biggest objective value increase.

8.0.1 By Dimension

1. Let $dim := 1$
2. If $dim > m$ stop. Optimal policy has been found.
3. Run the backward pass of *OPI*
4. If there is no increase in objective value for dimension dim, let $dim := dim + 1$ and continue with step 2.
5. Run the forward pass on dimension dim

6. Update the policy
7. Continue with step 3.

This strategy is simple. It uses all the available resources of the first dimension before continuing to the second, and so on. However, this may, and in many cases will, lead to a bigger number of iterations than absolutely necessary. If the total number of allocation "slots" is exceeded by the number of available units, the constraints giving rise to this situation will not be binding in the first dimensions that are "filled up". So if a store has very small profits compared to the others, but is chosen very early in this process, many, if not all, of its units will be allocated, only to be pushed out later. The probability of having one dimension with extremely low profits and very high capacity, which is filled up early and will create a very big overhead in terms of iterations could be avoided by choosing the dimensions not in ascending but in a randomized order.

8.0.2 Best Increment

1. Run the backward pass of *OPI*.
2. If the maximum increase of objective value is 0, stop. Optimal policy has been found.
3. let *dim* be one dimension with maximum increase of objective value.
4. Run the forward pass on dimension *dim*.
5. Update the policy.
6. Continue with step 1.

This strategy is, in concept, a bit more sophisticated than increment *by dimension*. It neatly avoids the problem of a dimension with inferior objective value contribution generating useless iterations by allocating units ordered by the increase of total objective value. Units of a dimension with very small objective value contributions would only be allocated very rarely, resulting in less iterations.

9 Implementation

The algorithms used to solve *CCCMAP* were implemented using the *Java* programming language version 1.5. Building was performed with the automatic build features of *Eclipse* version 3.2 using the *JDT (Java Development Tools)*.

9.1 Storing Problem Instances

Problem Instances are stored in text files as described in Appendix A. A class called *Instance* reads and writes those files, provides access to the data via getter and setter methods, and can create random instances for testing.

9.2 Standard DP

Simple Pseudo-code for this algorithm looks like this:

1. Initialize prizes for time T
2. For every possible configuration at time $T-1$, visit all possible states and in each one of them, check all possible state transitions to states of time T. The value of a state at time $T-1$ is the best sum of sale profit at time $T-1$ and remaining value at time T encountered in legal (w.r.t. constraints) transitions. For each configuration, remember which state transition led to its objective value, by saving the index of the corresponding configuration at the next time step (this configuration will be called "next configuration" for brevity).
3. Repeat for steps $T-2 \ldots 0$ (*backward pass*).
4. Look for the configuration with the best objective value at time 0.
5. Using the references to the next configuration, reconstruct the optimal policy (*forward pass*).

Looking at steps 2 and 3, it becomes obvious that, for each configuration, we have to store both an objective value and a reference to the corresponding configuration at the following time step. Since the configurations are an m-dimensional domain, we need multidimensional arrays for both.

Since the objective values are only used in the backward pass (and then only by the following step), it is possible to store just the data for the current and the previous step. The ideal data structure would be two m-dimensional arrays, with the number of units of each dimension as this dimension's index.

The forward pass uses the references to the next configuration. Hence, all these references have to be stored. For one time step the data is m-dimensional, so a natural way would be adding one dimension for time.

For the algorithm to be able to deal with input of different dimensionality, we need a way to store multidimensional arrays where the number of dimensions can be chosen at runtime. Since this is not a property of Java container classes and no useful libraries were found for this, the necessary infrastructure was implemented to store the data in one-dimensional arrays and convert their indices to indices of appropriate dimensionality and vice versa.

The whole DP-functionality was implemented in a monolithic class called *DPSolver*, since coupling would have been very strong in an implementation using more than one class. It uses the *Instance*-class to load and store data.

9.3 Optimal Policy Iteration

As described in chapter 6, *OPI* incrementally uses *Dynamic Programming* to incrementally find an optimal solution to *CCCMAP*. In every iteration, Dynamic Programming is performed in a restricted part of the state space around an optimal solution of a "smaller" problem instance, eventually leading to an optimal solution to the instance one is trying to solve.

For actually performing this DP-step, the optimal solution to the previous problem instance is projected into a new state space, in which its configurations map to the value 1 in all dimensions and at all time steps. We perform the backward pass of dynamic programming in this new search space, allowing state values from 0 to 2 in each dimension, with 0 meaning "one unit less than in the previous solution", 1 meaning "the same" and 2 meaning "one unit more". To see whether a state transition in the transformed search space is legal, it is necessary to transform the values of its starting- and end-point back to the original state space.

Assume we have an optimal policy for an intermediate problem instance with n dimensions. Then the search space we are using has $n + 1$ dimensions, n for the stores (with size 3 each) and one for time (with size m). This is why, like with the standard DP-algorithm, the dimensionality of the search space is variable. Therefore we work with multi-dimensional indices for one-dimensional arrays just as we did in the standard DP-approach.

Let us consider the data structures we need to perform one OPI-iteration: For the backward pass, to get the value of a configuration at time t, we need

- the value of all configurations at time $t + 1$ as end-points of the state transitions and
- the configuration of the old optimal policy at times t and $t + 1$ as offsets, to be able to check whether a state transition is legal.

A state transition is legal if its starting- and end-point have non-negative value and the allocation represented by it is legal w.r.t. the constraints. In every step of the backward pass, every configuration is considered n times a starting- or an end-point. To check its legality, it must first be transformed back to the original state space. To reduce the computing time required for such transformations, it makes sense to use two Boolean arrays, one for all configurations at time t and one for time $t + 1$, in which we store the pre-computed values of whether these points can be part of a legal state transition, requiring only one transformation per configuration. For the forward pass we need some kind of pointer for every configuration in our search space, which points to the configuration at the next time step which was used to calculate its best value in the backward pass.

Because of these considerations, the implementation done for this thesis uses the following data structures:

- An integer array of size $T \times m$ for storing the configurations of the old optimal policy (called *old*)

- Two *double*-arrays of size 3^m to store the values of the configurations at the current and next time step during the backward pass (*valHere*, *valNext*).
- Two Boolean arrays of size 3^m that store whether a configuration is a legal starting- or end-point of a state transition (*legalStart*,*legalEnd*).
- One array of integers of size $3^m \times t$ to store the "pointers" for the forward pass. The pointers are implemented as integers, which are converted to the m-dimensional index of the corresponding configuration at the next time step (*pNext*).

To initialize the data structures, we only need to set the values of all configurations in the *old* array to 0, which is the trivial solution to the problem instance with all capacities set to 0.

Given an optimal policy to a sub-problem-instance, the backward pass of OPI is executed as follows.

- Initialize *valNext* with zeros (as we define the value of any configuration at time t to be zero.
- For every time step t in $T-1,\ldots,0$:
  - Initialize the *legalStart* and *legalEnd* arrays by checking whether the configurations are non-negative in every dimension.
  - Initialize the *valHere*-array with minus infinity, such that every legal transition creates a better value.
  - For each legal configuration at time t check which of the transitions to legal configurations at time $t+1$ is within the constraints by transforming it back to the original state space. Of the legal state transitions, choose the one that yields the best value for the current configuration, write this value to *valHere* and a reference to the corresponding configuration at $t+1$ into *pNext*.

For the forward pass, we first have to choose which trajectory to follow. When using *best increment*, we look at all configurations at time 0 that are bigger than the starting point of the old trajectory in at most one dimension, and choose the one with the best objective value. If there is no increase in objective value, we terminate the algorithm and output the previous trajectory as the optimal solution. For *increment by dimension* we check whether a capacity increase in current dimension improves the objective function and follow its trajectory if it does. If there is no increase and the current dimension was the last dimension, we terminate. Otherwise the next dimension becomes the current dimension and we do the same checks again.

Best increment and *increment by dimension* are implemented in one *Java*-class each, using the *instance*-class for loading and saving problem instances from files.

9.4 Evolutionary Algorithm

For the purpose of comparison, an *Evolutionary Algorithm (EA)* was implemented. The aim was to see how an EA with low implementation complexity

will perform compared to OPI, with respect to both running time and quality of results.

Evolutionary Algorithms are inspired by Darwinian Evolution. To find a solution to a problem with a good objective value, one simulates the effects of selection, recombination and mutation on a population of solutions until certain termination criteria are met.

A common technique in EAs is to relax some of the problem's constraints and encourage valid solutions by penalizing constraint violations in the objective function. This is very similar to *Lagrangean Relaxation* in Linear Programming. This potentially makes it possible to find solutions that would not have been found using only valid solutions in the population. Furthermore, one can use simpler mutation and recombination operators and wait for the individuals to converge toward valid solutions. A drawback is that there is no guarantee that solutions found this way are valid.

A very important decision is how strongly to penalize constraint violations. One wants to encourage valid solutions but keep the positive effect of a more diverse population. Using a fixed penalization scheme is one possibility. The other is to adapt parameters that determine the penalties. The latter is called *Stepwise Adaption of Weights (SAW)*, which was chosen for this implementation.

The characteristics of this implementation are:

- *Individuals:* Not necessarily valid policies for the problem.
- *Initialization:* The policies are generated at random. To do this, the available capacities of each store are randomly allocated across the time steps, without trying to meet the allocation constraints.
- *Mutation:* One individual is cloned. Then, at random, either one or five times, one decision variable of the new individual is altered by adding or subtracting a random number. Each of the two variants is used to generate 100 individuals per generation.
- *Recombination:* Two individuals, A and B are chosen at random from the population and a new individual is created based upon both: The decision variables for some stores come from A, the others come from B, with the assignment being random. This is done 100 times per generation.
- *Selection:* A simple tournament selection is used: Until the population has the desired size (set to 200 individuals), two individuals are picked randomly and the one with lower fitness is removed from the population.
- *Weights:* The penalization is initialized as 1 % of the best profit in the whole problem for each violation of either capacities or allocation limits, and, after 10 iterations without objective value gain, increased by 1 % of the best profit.
- *Termination:* The algorithm terminates after the penalties are set to at least 1.5 times the value of the best allocation profit and at least 300 iterations have passed without profit increase. Note that for a penalty bigger than the best profit, each removal of a unit in violation of a constraint improves the objective function value.

10 Computational Experiments

This section is an overview over the experiments performed during the practical part of this thesis project. All experiments were performed on a computer with an *Intel Core Duo* processor with a clock speed of $2.16GHz$ and 2 Gigabytes of RAM running *MacOS X* 10.5.2. Note that the implementation is not optimized for parallel processing and makes use of only one processor core.

The instances uses for the tests were generated randomly. For given dimensionality and allocation limits, the profits are created the following way for each store and time step: The profit of the first allocation is the product of two pseudorandom floating point numbers between 0 and 1 generated by *Java's Random*-Class. For all n greater than one and smaller than the corresponding allocation limit, the profit of the n^{th} allocation is the product of the $n-1^{st}$ allocation multiplied by a psedorandom number between 0 and 1. This scheme guarantees non-increasing profits.

10.1 Optimal Policy Iteration

First we look at the implementation of Optimal Policy Iteration, trying to find at the difference in running time between the two possible ways of incrementing capacities.

10.1.1 Order of OPI Iterations

As described in chapter 8, *Best Increment* minimizes the number of iterations, while *Increment by dimension* may allocate units that may be de-allocated in later iterations, generating excess iterations. We will be using an example with four dimensions, 30 units per dimension, 20 time steps and an allocation limit of 4 units per time step to visually illustrate these differences.

Figure 18 visualizes the capacities of the stores over all iterations needed by *Best Increment* to reach the optimal solution. Each dimension's capacity is plotted in a different color. The plot on the right hand side shows how the objective value changes over the iterations. Obviously, the profit of the increments decreases, until no more profit can be made by incrementing capacities and the optimal solution is reached.

Figure 19 shows the capacity increases that lead to the same optimal solution when running *Increment by Dimension* with different dimensions plotted in different colors. The fact that the capacity increases are ordered is clearly reflected in the plot. We can see that the first two dimensions are filled to their full capacity. But when the total number of possible allocations, determined by the cross-constraints, is reached, some units need to be removed to enable the allocation of more profitable units from other dimension. Each of the units removed has obviously been allocated before, so every unit that is removed corresponds to one redundant iteration. The right hand side plot shows the objective values corresponding to the iterations. We can see that the profits of increments are non-increasing while the same dimension is increased, but changing the dimension leads to "bumps".

The fact that *Increment by Dimension* can lead to redundant iterations (also reflected in the number of iterations in figures 18 and 19, leads to the intuition that *best increment* should have significantly better performance. To

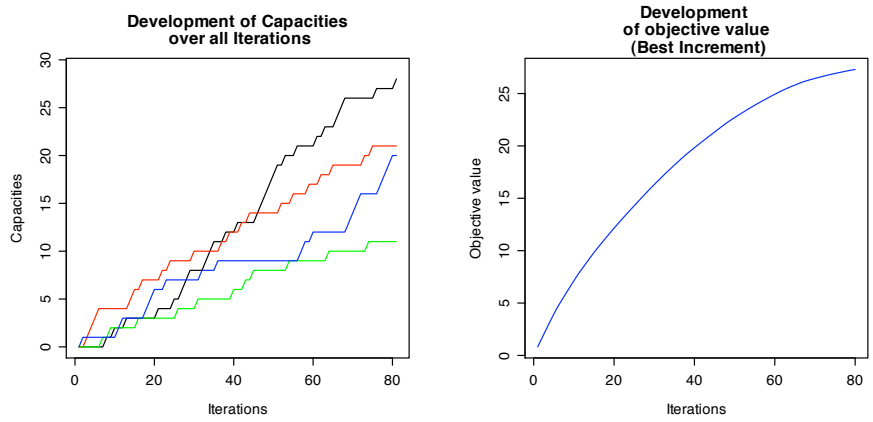

Figure 18: Development of capacities and objective value for *Best Increment*

the surprise of the author, the case was not that clear. *Increment by dimension* even turned out to be superior in some instances of higher dimensionality.

After some analysis of the implementation's source code, the reason became apparent. When there have been no increments in a store yet, i.e. its capacity is still set to 0, its state variables are all 0 at all points in time. Figure 20 illustrates the search-tube in such a situation with a two-dimensional example.

This means that in this dimension, a third of the search tube of the backward pass contains state values. This part of the search tube cannot contain the trajectory because state variables cannot be less than 0.

When calculating the values of state transitions in the backward pass, all transitions that start with a negative state value ($\frac{1}{3}$ of the transitions) and/or end with a negative state value (again, $\frac{1}{3}$ of the transitions) will not be considered. In total, this means that only for $\frac{2}{3} \cdot \frac{2}{3} = \frac{4}{9}$ of the state transitions, their objective values need to be computed. This computation involves two memory lookups, namely the value of the end-point of the state transition and the profit of the actual transition, which is a computationally intensive aspect of the backward pass. Hence, reducing the number of these computations significantly speeds up the entire backward pass.

This implies that, when using *increment by dimension*, having dimensions which are cheaper to increment may or may not compensate, and in some cases even over-compensate, the increased number of dimensions. The following examples were randomly generated, with an increasing number of dimensions and 5 time steps. For n available units, the number of possible allocations was set to $\lfloor \frac{1}{2}n \rfloor + 1$. On some instances *increment by dimension* performed better, on others *best increment* was faster. Table 4 shows the running times and figure 21 shows the a graphical comparison.

The examples did not show benefits for one or another method. However, one can construct examples where *increment by dimension* creates a large number of unnecessary iterations. Consider an instance with only one time step, a total allocation limit of n units and m dimensions and the profits given in table 5.

The optimal solution is obviously allocating only the n units from store m.

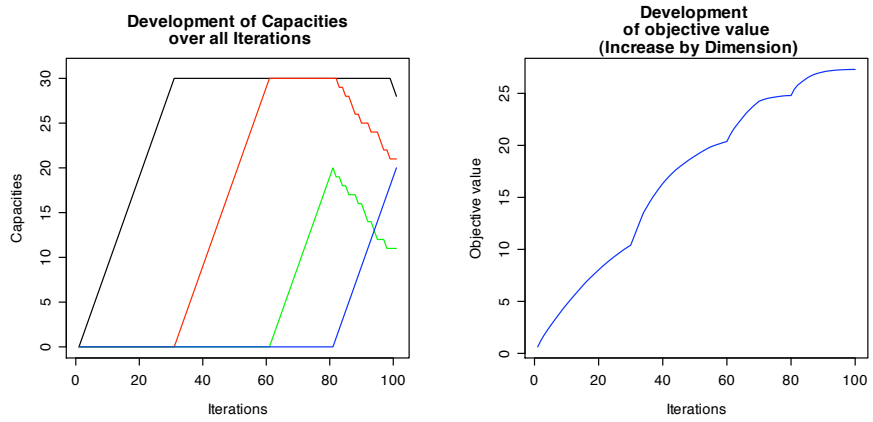

Figure 19: Development of capacities and objective value for *Increment by Dimension*

Table 4: Running times of *Increment by Dimension* vs. *Best Increment*

# of Dimensions	Running time IBD (ms)	Running Time BI (ms)
1	4	4
2	10	7
3	19	20
4	156	50
5	505	692
6	5706	3167
7	16444	21920
8	281204	298394

While *best increment* would need n iterations, choosing only the units from store m, *increment by dimension* will first choose the units from store 1, then dimension 2 and arrive at the optimal solution after $m \cdot n$ iterations.

This shows that *increment by dimension* may, but need not be faster than *best increment*, depending mainly on the problem instance. Since we know that the number of iterations will always be minimal for *best increment*, it seems to be the safer choice for any implementation.

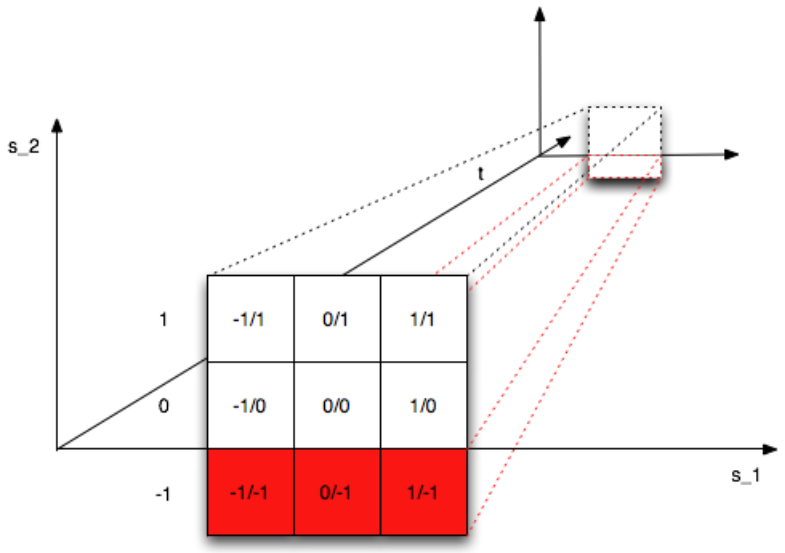

Figure 20: Two-dimensional example with a third of the search tube being irrelevant

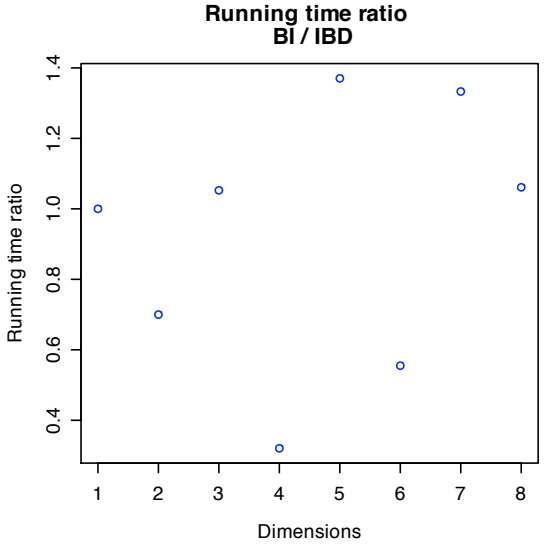

Figure 21: Ratio of running times

Table 5: A worst-case constellation for *Increment by Dimension*

x	$\delta_{01}(x)$	$\delta_{02}(x)$	...	δ_{0m}
1	x	$x+1$	...	$x+m-1$
2	x	$x+1$	...	$x+m-1$
...				
n	x	$x+1$	...	$x+m-1$

10.1.2 Cascade Effects

In chapter 7 we considered, looking at the network flow correspondence, how the optimal solution might change if we add one unit to a store. A new unit may be allocated replacing another unit, which will be allocated at another time step replacing yet another unit etc., which we call a *cascade effect*.

We will now use the instance from the last chapter to see them occurring when actually running *OPI*. The plots we will consider are visualizations of the output from running *increment by dimension*. While there are also cascade effects when using *best increment*, they are more common with *increment by dimension*, which makes the latter a better data source for visualizing the effects.

To understand the plots, first look at figure 22. It contains three plots. The first two plots show the configurations of a store over time in two consecutive iterations. The third plot shows the difference between the two plots, i.e. the change that was the result of an *OPI*-iteration taking place between them. It is the simplest change that can take place. One unit is allocated and nothing else happens.

After some iterations, the graphics plotting the configurations against time contain a lot of information in a very dense visual representation. This makes it hard to spot the difference between two of them. Hence, we will use the delta-plots to illustrate the changes made by *OPI*-iterations. Figure 23 shows a unit being allocated, replacing a unit from another dimension, which is then allocated at a later point in time. A unit that is replaced can also be allocated at an earlier time slot, as shown in figure 24.

A unit that is moved because of an allocation can also move another unit. This can be seen in figure 25, where the allocation of the new unit moves one unit forward, which in turn moves yet one more unit back.

When all the allocation-cross-constraints are exhausted, *increment by dimension* may allocate a unit that "pushes" another unit out of the optimal solution. This is shown in figure 26. This can also happen indirectly, as can be seen in figure 27.

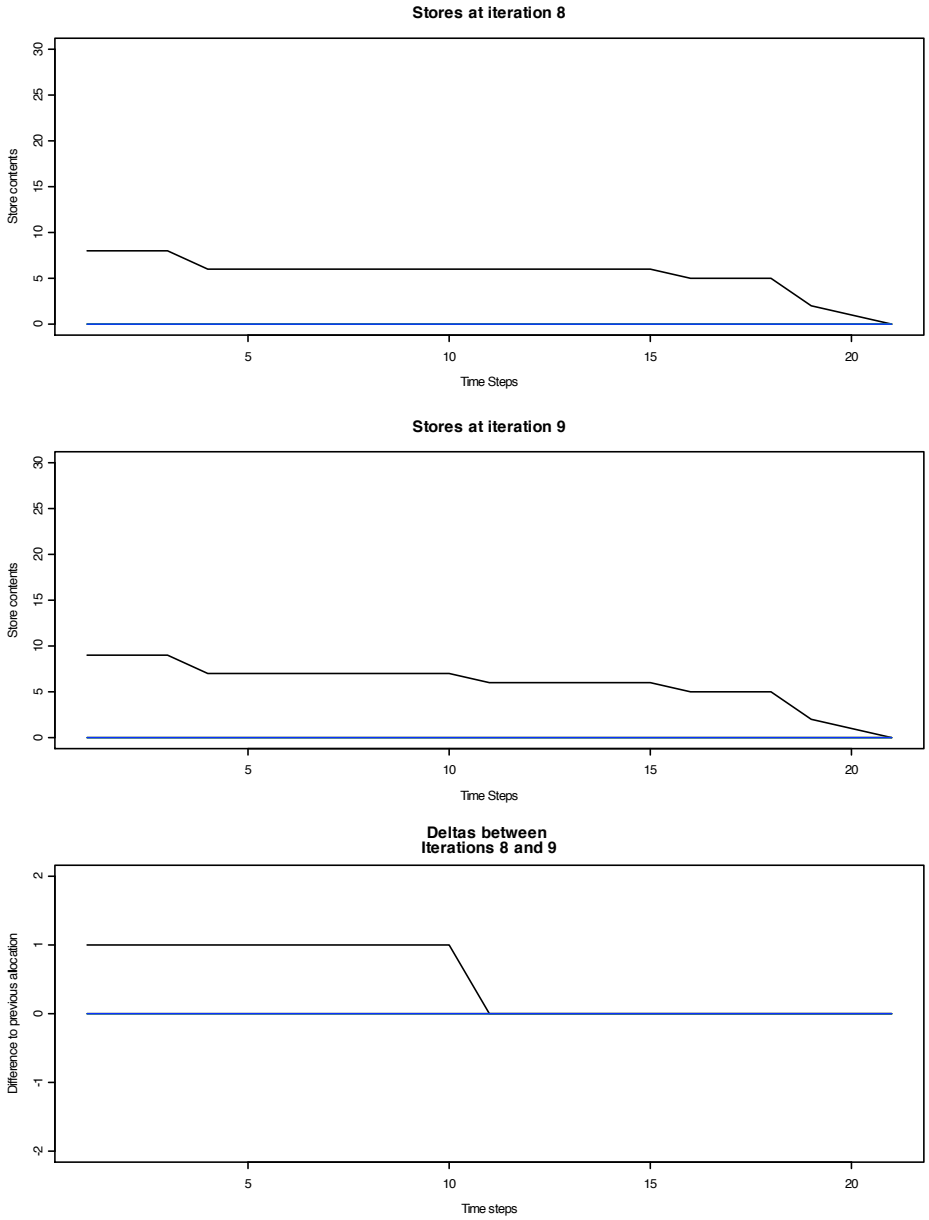

Figure 22: Change between two iterations.

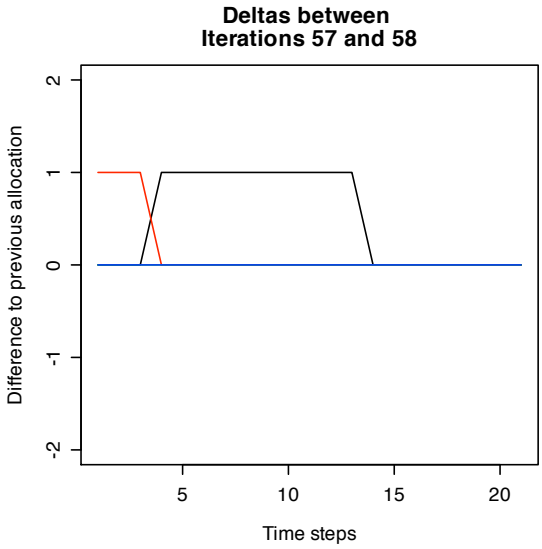

Figure 23: Adding one unit, replacing another unit which is allocated later.

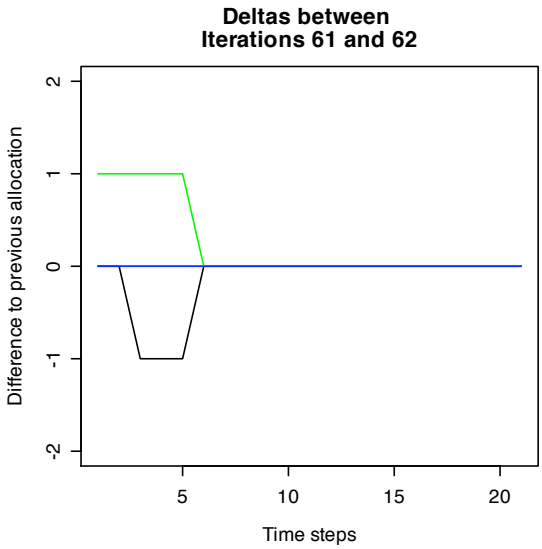

Figure 24: Adding one unit, replacing another unit which is allocated earlier.

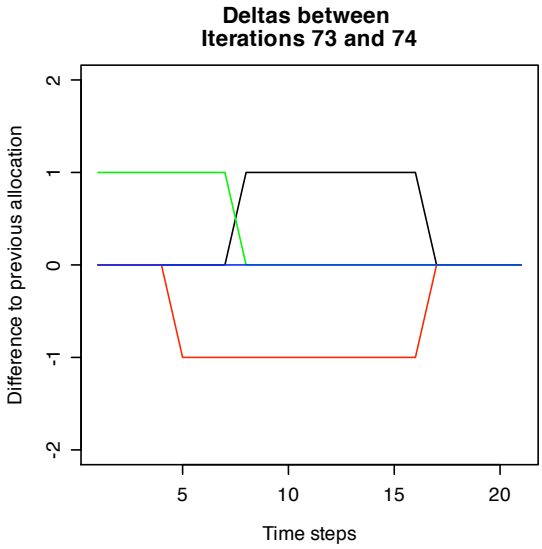

Figure 25: Another cascade effect.

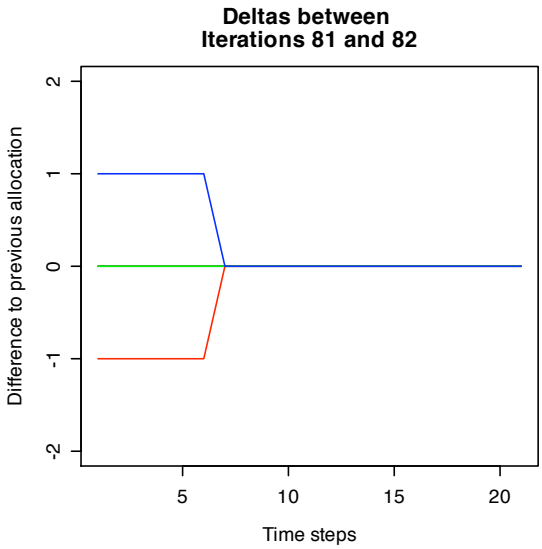

Figure 26: Adding one unit, replacing another unit which removed.

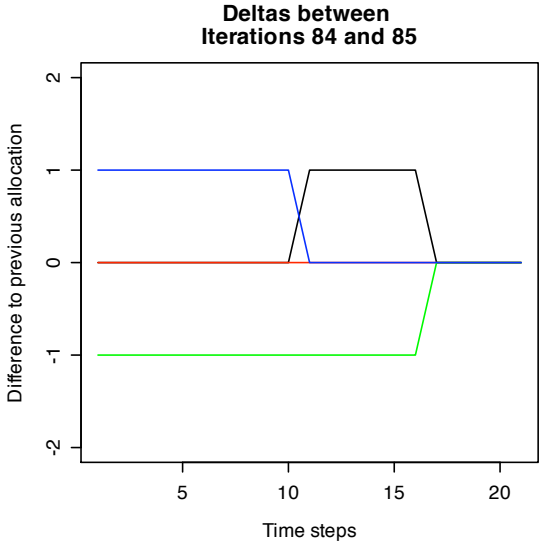

Figure 27: Pushing one unit out via a cascade effect.

10.2 Comparison of OPI and naive DP

The implementation of the simple DP algorithm allows us to compare the performance of *OPI* to simple DP's undesirable running times we were trying to improve. As indicated before, if the running time of DP for a problem instance with m dimensions and a capacity of s in each dimension is $O(x)$, then the running time for a problem instance with the same number of dimensions and capacities of ks in each dimension, is $O(x \cdot k^{2m})$. Such an increase should, however, only affect the running time of OPI by a factor of k.

Hence, the best way to show the performance improvement is to take a series of instances with a fixed number of dimensions but increasing capacities. For a practical test, 4 dimensions, 10 time steps and at most 10 allocations per time step were chosen. The sum of the capacities ranges from 1 to 28 units. The experiment compares the *Best Increment* strategy of OPI to the simple DP algorithm on these instances.

Table 6 shows the results of this test. As expected, the running time of naive DP increased dramatically, while the increase in running time of OPI was linearly bounded.

This data cannot be meaningfully compared on a linear scale. Figure 28 therefore compares the running times on a logarithmic scale. Figures 29 and 30 plot the running times of the two algorithms separately on linear scales.

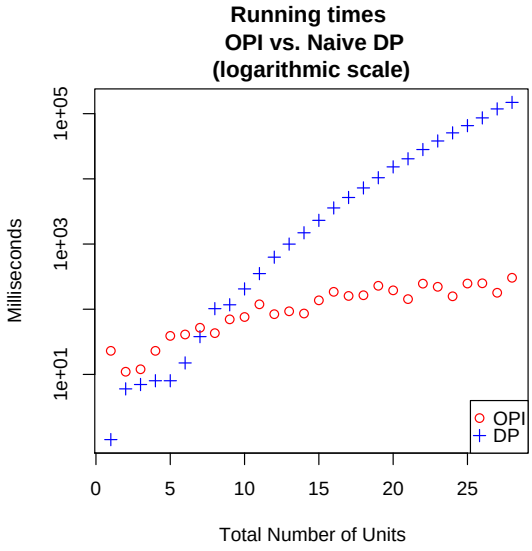

Figure 28: OPI and simple DP applied to 4-dimensional problem instances of increasing capacity

These results confirm that *OPI* successfully fixes the biggest performance problem the simple DP algorithm has when solving *CCCMAP* by turning the capacities of the stores into a linear factor.

Table 6: CPU-times of OPI vs. simple DP

# of units	OPI (ms)	naive DP (ms)
1	23	1
2	11	6
3	12	7
4	23	8
5	39	8
6	41	15
7	52	38
8	43	102
9	70	117
10	76	205
11	119	352
12	84	629
13	93	998
14	86	1489
15	137	2309
16	185	3580
17	160	5177
18	164	7259
19	229	10410
20	195	15272
21	143	20317
22	247	28177
23	221	38119
24	158	50760
25	248	65539
26	250	86174
27	179	118279
28	304	148940

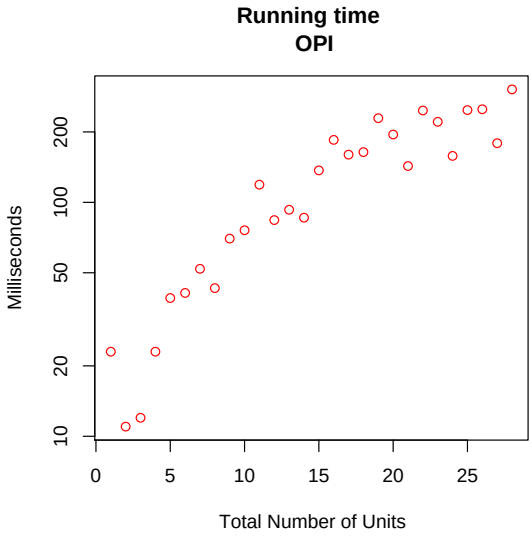

Figure 29: OPI performance on a decimal scale

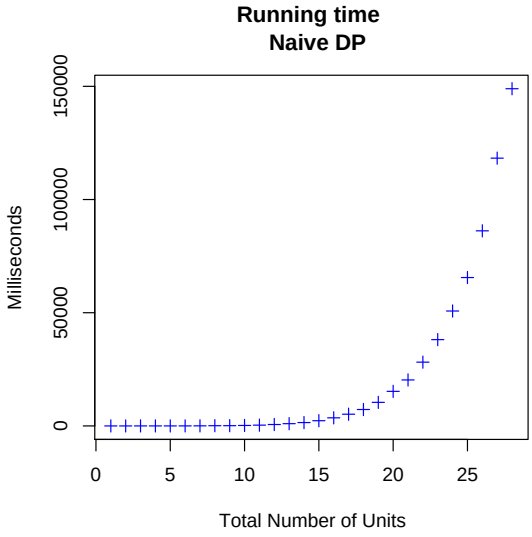

Figure 30: DP performance on a decimal scale

10.3 Comparison of OPI and the Evolutionary Algorithm

This chapter looks at how *Optimal Policy Iteration* performs compared to an *Evolutionary Algorithm (EA)*. The behavior of the *EA* itself is analyzed thoroughly before comparing it to *OPI*. In this chapter, all running times of the *EA* are the mean of 30 runs, rounded to milliseconds. Furthermore, the standard deviation is provided.

10.3.1 Performance Characteristics of the Evolutionary Algorithm

The mutation and crossover operators of the Evolutionary Algorithm implemented for this thesis change the decision variables in a randomized way. In other words, neither the result nor the running time is pre-determined for a given problem instance. Empirical tests are therefore the best method of determining the runtime behavior of the algorithm.

As described in chapter 9.4, the mutation and crossover operators do not guarantee that mutations or offspring of a legal policy are legal policies as well. However, constraint violations are penalized, i.e. the value of the profit function is reduced for every constraint violation. Those penalties are increased during the running time of the algorithm until every allocation that violates a constraint violation is penalized by a higher value than its profit can be. In the beginning, though, an allocation that violates a constraint, may yield a profit that is greater than the penalty for the corresponding violation (see chapter 9.4 for a detailed description of the parameters used in the implementation).

Hence, in the beginning we can expect to see more units allocated than available, and time steps where the time-specific allocation limits are exceeded. When the penalties are increased, the less profitable violations disappear, until a legal solution is reached. This can be seen clearly when looking at the plot in figure 31. It plots the EA's profit function against the iterations of one run of the algorithm on a problem instance with 4 dimensions, 30 units per dimension, 20 time steps and an allocation limit of 4 units per time step. Note that, in the end, when the penalties increase, the quality of the average solution decreases. This seems to be due to illegal solutions being less and less profitable, therefore taking up a decreasing part of the population.

Since the EA does not search the all possible legal policies, but rather a set of profitable policies (that are, in the beginning, not necessarily legal), the number of possible configurations should be of little impact. One would expect other factors to be more important: The number of decision variables, the total number of units in the problem instance, or the number of possible legal allocations.

First we will check whether the number of dimensions is relevant to the performance of the EA. To do this, we are running the EA on a series of instances with increasing number of decision variables. For each number d of decision variables we create two instances. One with 10 time steps, $\frac{d}{10}$ dimensions, each with capacity 20 and an allocation limit of $\frac{2 \cdot d}{10}$ units per time step, and one with 5 time steps, $\frac{2 \cdot d}{10}$ dimensions, a capacity of 10 units per dimension and a limit of $\frac{4 \cdot d}{10}$ units per time step. This means that both instances have the same number of units available, the same total allocation limit and the same number of decision variables, but a different number of dimensions. The results of this experiment can be viewed in table 7 and figure 32 do not indicate that

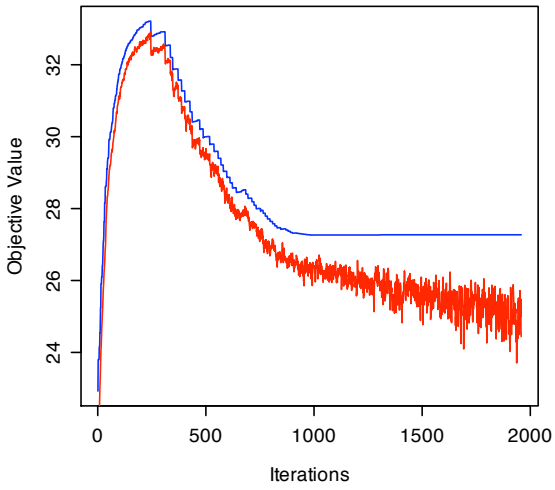

Figure 31: Objective function of EA, best (blue) vs. average value (red)

the number of dimensions influences the running time of the EA. What we see clearly is the linear impact of the number of decision variables on the running time.

Table 7: Impact of Dimensionality

# of dec. vars.	fewer dims. (ms)	std. dev.	more dims. (ms)	std. dev.
10	3600	130.9683	3253	101.77045
20	5534	212.2356	4579	58.34814
30	6398	131.1239	6135	203.21837
40	8223	251.6215	7584	141.76570
50	9737	310.8023	9123	721.69024
60	11677	268.1619	10268	401.50609
70	12471	261.4463	13478	412.39063

To check the influence of whether all units can be allocated (we call this a *saturated instance*) or not (*unsaturated instance*), we created instances from 1 to 7 dimensions, with 10 time steps, 20 and units per dimension. To get an unsaturated instance, we set the allocation limit to twice the number of dimensions per time step, and half of that for a saturated one. The results are summarized in table!8 and visualized in figure 33. Saturation seems not to influence the running time.

The last experiment checks how big of a difference the number of units per dimension makes. To do that, we created instances ranging from 1 to 7 dimensions, with 10 time steps. For a dimensionality of d we chose either 20

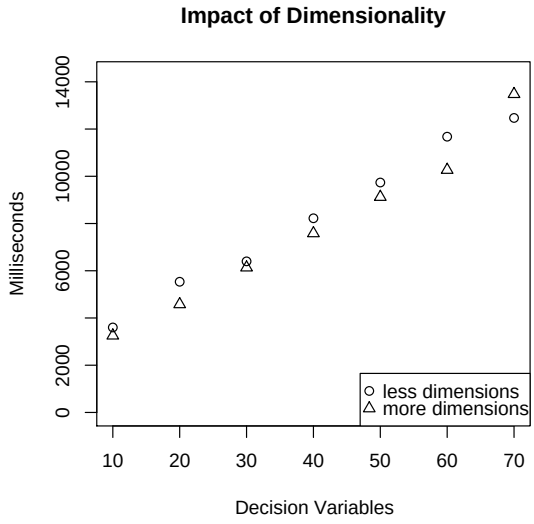

Figure 32: Impact of dimensionality on the running time of the EA

Table 8: Impact of Saturation

# of dec. vars.	unsaturated (ms)	std. dev.	saturated (ms)	std. dev.
10	3562	99.29302	3384	69.31193
20	5138	130.13980	4532	55.12865
30	6604	92.22101	5767	110.56231
40	8507	193.15643	7039	97.77974
50	9939	239.39156	8560	164.82319
60	11960	640.29286	10721	279.23318
70	12480	462.76025	13821	328.59800

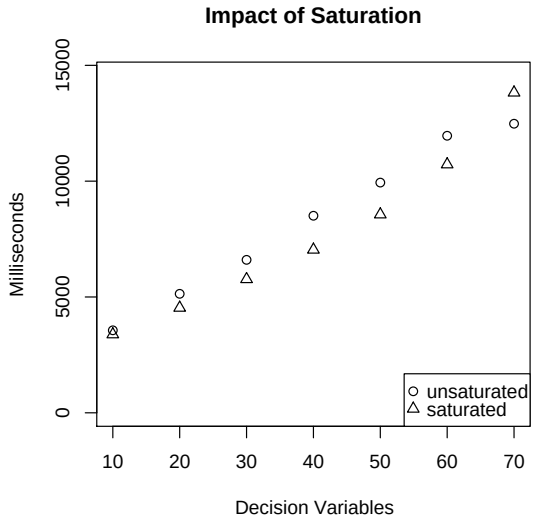

Figure 33: Impact of saturation on the running time of the EA

or 40 units per dimension, and allocation limits of either $2 \cdot d$ or $4 \cdot t$ units per time step and dimension. Table 9 and figure 34 show the results. The total number of units seems to affect the running time as a linear factor with a small coefficient.

Table 9: Impact of Capacities

# of dec. vars.	less cap. (ms)	std. dev.	more cap. (ms)	std. dev.
10	4153	196.0169	3743	195.66394
20	6039	276.7526	5246	73.43103
30	8303	858.1097	6748	202.73657
40	10805	796.3802	8356	442.84328
50	12405	974.2088	10645	461.54626
60	14795	1362.1322	11178	270.57560
70	17245	1766.4727	12942	628.31907

The experiments showed that the number of dimensions indeed does not matter, that the number of decision variables is the most important (linear) factor. The number of possible allocations has little impact on the running time, but saturated instances have more predictable running time. The sum of the capacities seems to influence the performance as a linear factor with small coefficients.

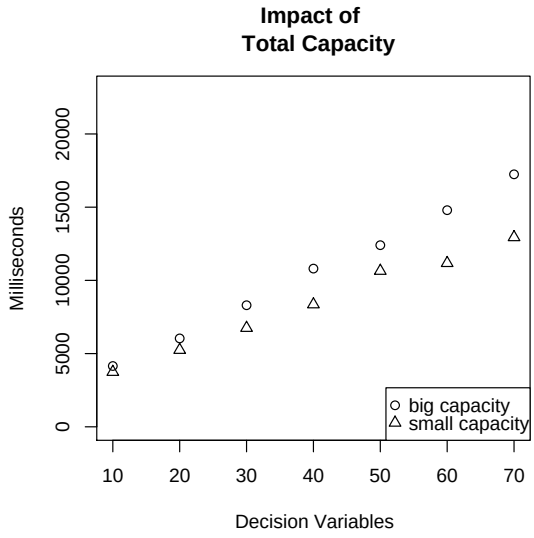

Figure 34: Impact of the total number of capacities on the running time of the EA

10.3.2 Comparison to OPI

As seen in the previous section, the most important factor for the EAs running time is the number of decision variables. For *OPI*, the most significant impact on running time comes from the number of dimensions. Hence, the best way to show the differences in performance between *OPI* and the EA is to use a series of instances with increasing dimensionality. To do this, we chose examples with 1 to 7 dimensions, 5 units per dimension, 30 time steps and an allocation limit of 1 unit per time step. Table 10 shows the results. *Quality* denotes the EA's average best solution divided by the optimal solution (computed using *OPI*, and measures how good the EA approximates the optimal solution. As expected, the running times of *OPI* increases exponentially, while the running times of the EA increases linearly.

To get a meaningful plot that compares these results, one must use a logarithmic scale, as done in figure 35. Figures 37 and 36 show the running times plotted against separate linear scales.

Table 10: Performance of EA vs. OPI

dims	OPI (ms)	EA (ms)	std. deviation	quality (EA)	std. deviation
1	20	8433	565.0908	1.0000000	0.000000000
2	26	12012	174.2651	0.9934545	0.001236241
3	85	15094	376.7782	0.9994806	0.002844628
4	967	19907	699.1662	0.9969838	0.005588005
5	10022	23602	877.6363	0.9888670	0.004086761
6	83673	27647	632.5258	0.9829171	0.007579513
7	912853	33659	1840.7905	0.9867220	0.007779252

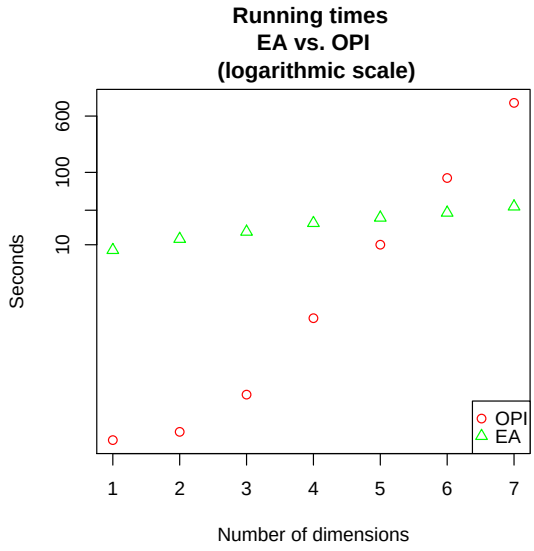

Figure 35: EA vs. OPI performance on an increasing number of dimensions

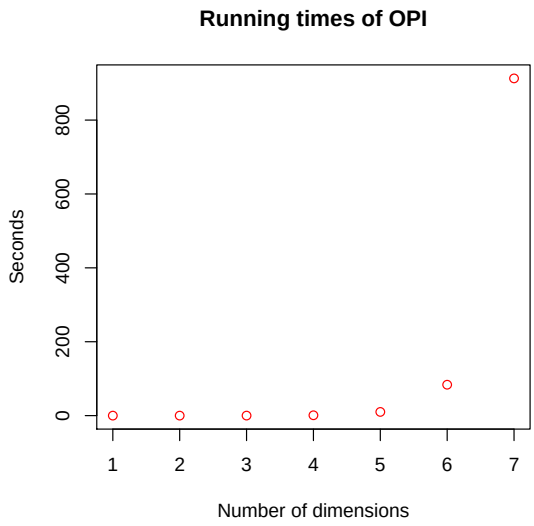

Figure 36: OPI performance on a decimal scale

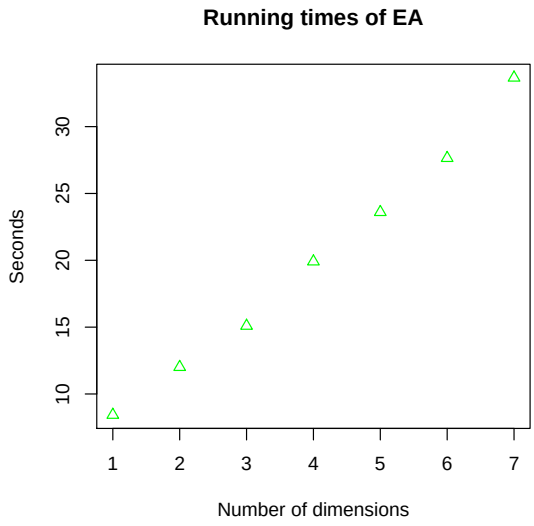

Figure 37: EA performance on a decimal scale

11 Limitations

The objective function value of *CCCMAP* represents prices for a good. It uses an expected development of prices for planning. For real-life applications, it would make sense to consider several pricing scenarios, which can be represented by probability trees.

When this thesis project started, it was thought that extending *OPI* to solve problems with stochastic pricing would be simple. However, while trying to find a proof that this was indeed the case, the author found a counterexample that shows that the neighborhood properties needed by *OPI* do not hold for stochastic pricing. This chapter explains this limitation in detail.

11.1 Deterministic Pricing

OPI works very well for deterministic pricing, since changes in the optimal solution created by adding one unit can only have an absolute value of ≤ 1 in every decision variable. The reason is, intuitively speaking, that adding one unit can, as described before, only cause a simple "cascade" of reallocation that is without cycles, i.e. no time step or store is affected more than once.

11.2 What Changes with the Introduction of Scenario Trees

In a stochastic setting, i.e with a scenario tree for pricing, units from different dimensions can contribute "simultaneously" (that is, at the same time step but in different scenarios) to the profit of one time step, even if they cannot all be allocated at the same time in one scenario.

Let us give an example to illustrate this. Figure 38 is a scenario tree that branches once. Let us first have a look at the composition of the figure, as it contains a lot of information and is also used in more complex examples.

- The first column is used to label the discrete time steps (from top to bottom), which shows us that this example has two decision points.
- The second column contains the maximum number of units that may be allocated at the given time step, called $fmax_t$ in the formal problem definition.
- To the right we see a scenario tree.
  - Each node contains, for each of the store, the profit $\delta_{t,j}$ that will be realized by selling one item from the corresponding store at the given time in this scenario (i.e. the pricing function is defined $v_{t,j}(x) = x \cdot \delta_{t,j}$ in the examples for this chapter).
  - Every branch is represented by several edges labeled with the probability of their occurrence.
  - One scenario is a path from the root node to a leaf node.

A policy, as shown in figure 39, is again a tree, which is isomorphic to the scenario tree. For each node in the scenario tree it contains a node with allocation decisions for each decision variable corresponding to the current time step. Note that we are considering decision variables, not state variables here.

Each decision has a value, and the sum of those values are also included in the corresponding table.

The total expected value of a policy can be computed in at least two ways:

1. By scenario: For each scenario, add up the value of the allocation decisions in the nodes on the path corresponding to the scenario, and multiply them by the probability of the scenario. The probability is the product of the probabilities along the edges of the path.

2. By node: For each node, add up the value of the allocation decisions in this node, multiply it with the probability of this node being reached in a scenario. This probability is 1 for the root and the product of the probabilities on the path from the root to the node for all other nodes. Then add up the values of the nodes.

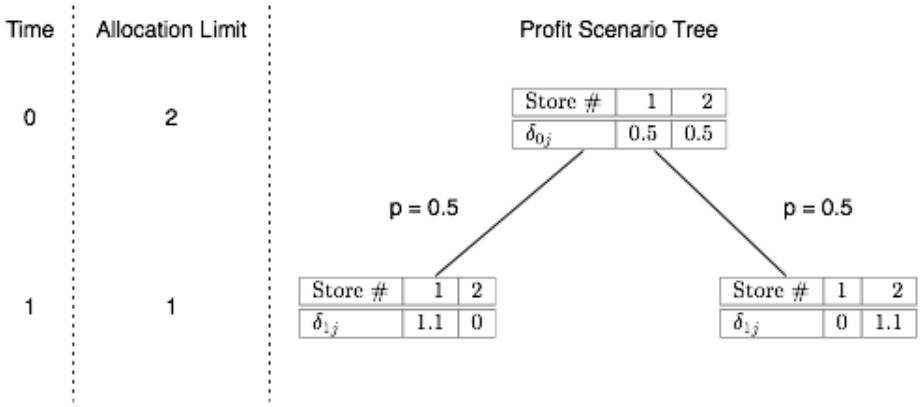

Figure 38: A simple scenario tree

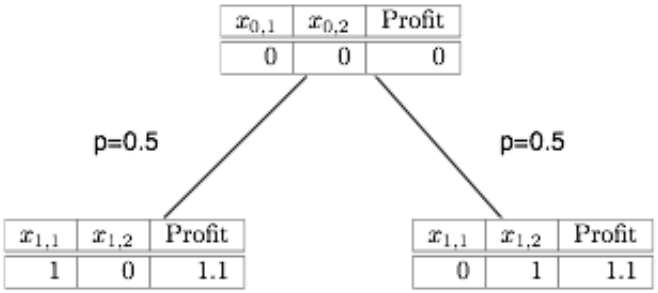

Figure 39: Optimal solution to figure 38 with capacities $< 1, 1 >$ yielding objective value: 1.1

Now assume we want to optimally allocate 1 unit each from store 1 and 2 given the scenario tree in figure 38. The best expected value is realized when we do not sell any unit at time step 0 and, for each of the scenarios, the more valuable unit at time 1 (which is what figure 39 stands for) yielding a total profit of 1.1. Obviously, because we have two independent scenarios, different units can be allocated in different scenarios. In this case this has the effect that even though only one unit can be allocated at time step 1, the allocation of two different units at that time contributes to the objective function. This property will be important later.

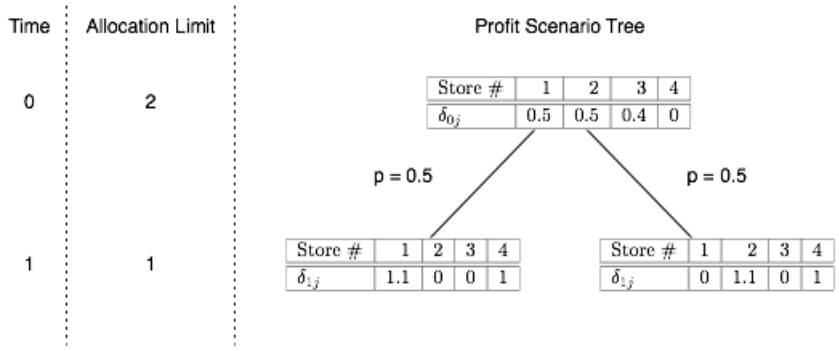

Figure 40: An example with 4 dimensions and 2 time steps

.

Consider figure 40, which uses the same idea as figure 38. Given 1 unit each in store 1 and 2 and2 units in store 3, the optimal policy is to allocate 2 units from store 3 at time 0 (profit: 0.8), and in each scenario, the better of unit 1 and 2 at time 1 (profit 1.1 in each scenario with $p = 0.5$) yielding an expected total profit of 1.9. This is illustrated in figure 41

However, when adding one unit in store 4, the optimal solution changes: The unit from store 4 will realize an expected profit of 1 when allocated at time 1, and no profit otherwise. The optimal solution here is to allocate one unit from

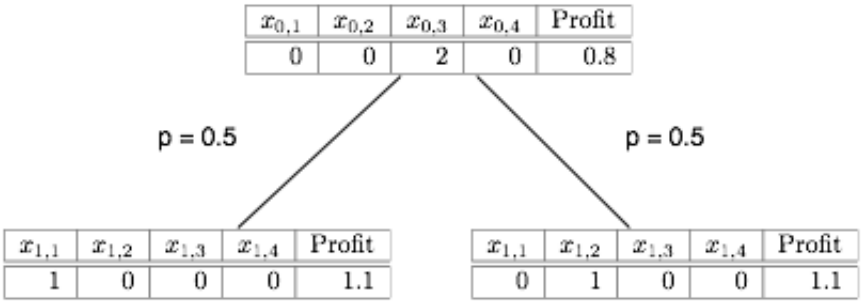

Figure 41: Optimal allocations for the problem from figure 40 with capacities $< 1, 1, 2, 0 >$ yielding objective value 1.9

both 1 and 2 at time 0 (profit: 1), one unit from 4 at time 1 in both scenarios (profit: 1 in each scenario with $p = 0.5$), yielding a total expected profit of 2, as illustrated in figure 42

Comparing figure 41 and 42 shows that the decision variables of store 3 have changed by a value of 2 at time 0, when there was a capacity increase of only 1 unit in store 4. This directly is due to the nature of the stochastic scenario tree.

In figure 41, one unit from either store 1 or 2 was allocated at time 1, depending on the scenario. Even though only one unit can be (and was) allocated at time 1 in any given scenario, both allocations contribute to the objective value, weighted by the probability of their nodes. Also, since the decisions before the scenario branch have to be compatible with both scenarios, both units are kept in time step 0.

Adding one unit in dimension 4 changes the situation: The profit from allocating it at time step 1 is bigger than what is lost when reallocating the units from store 1 and 2 to time step 0, replacing both units from store 3 due to cross constraints. This is a direct effect of stochasticity: With deterministic pricing, we used network flows and their property of skew symmetry, to show that, in a cascade of replacements, one unit can only replace one unit, and changes of more than 1 in any decision variable yield a contradiction. In the stochastic setting, the flow correspondence will not work, since the scenarios are different “threads of reality” , something that cannot be modeled in a flow network like the one we used in the proof.

Independent of the likelihood of such a constellation in a real-world example, this shows that Optimal Policy Iteration is unsuitable for optimizing the given formal problem when pricing is subject to a scenario tree.

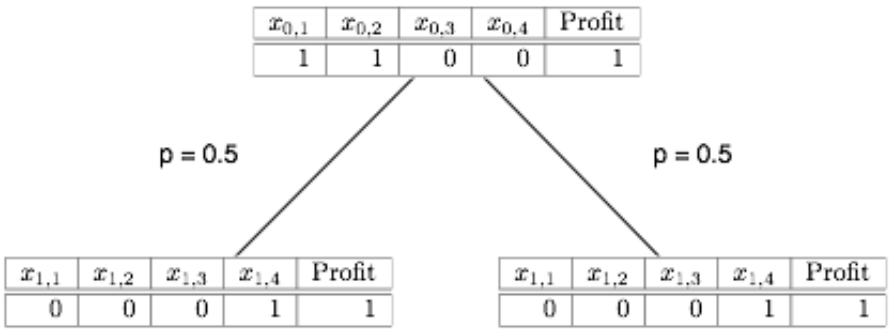

Figure 42: Optimal allocations for the problem from figure 40 with capacities $< 1, 1, 2, 1 >$ yielding objective value 2

12 Conclusions

In this thesis we introduced an incremental *Dynamic Programming* approach called *Optimal Policy Iteration (OPI)* for solving *CCCMAP*, an allocation problem with several stores and discrete time steps, varying prices over time and cross-constraints over the stores. We are trying to find an *Allocation Policy* that satisfies all constraints and is optimal with respect to profit.

The formal problem definition is inspired by the task of optimally using kinetic energy, stored in hydroelectric power plants in the form of water reservoirs, to create electricity at the most profitable times. Alfred Kalliauer, who works in the field of power trading, devised *OPI* as a solution to this problem.

We proved a neighborhood property between optimal solutions of problem instances that are identical, except for a difference of one unit in one store. Given an optimal policy for a problem instance, this property allows us to find the optimal policy for any instance that differs by one unit in the capacity of one store with an efficient DDP-search. This search consists of two passes:

- A backward pass that searches a neighborhood of the optimal solution to the current sub-instance. This pass results in the values of the optimal policies for all the sub-instances which are similar to the current one in the way described before, as well as the data necessary to compute the policies.
- A forward pass that computes the actual policy for one of these instances based on data from the backward pass.

To turn this into a usable algorithm for actually solving more complex instances of *CCCMAP*, we start with a trivial instance, which has the same constraints as the one we are trying to solve, except for all capacities, which are set to zero. This trivial instance has an equally trivial optimal solution, which is not allocating any units. Starting from this solution, we can add to the capacities of the stores, one unit at a time and calculate the optimal solution to this sub-problem-instance using DDP. We call the process of adding one unit to the capacity of a store and finding its optimal solution via DDP an *iteration* We have shown that an optimal solution to our original problem has been found if we either reached the capacity constraints or adding more units does not yield an increase in profit in any dimension.

In other words, the algorithm calculates the optimal policy for a problem instance via iterating over a series of optimal policies for smaller sub-problem-instances. This is the reason for its name, *Optimal Policy Iteration*.

With respect to performance, the advantage of *OPI* is being able to separate the impact of the number of dimensions and the capacities of the stores on the running time. The worst case running time of a single iteration depends only on the number of dimensions and time steps, whereas the number of iterations depends on the total number of units in all stores. This is a big improvement compared with a simple DP approach, in which doubling the number of available units in each store increased the worst case running time by a factor of 2^{2m} where m is the number of dimensions, whereas the same change only doubles the running time of OPI.

The order in which the capacities are increased can be chosen freely. In this thesis, we looked at two ways of systematically doing so: *Increment By*

Dimension (IBD), which increases the capacities one store after another, and *Best Increment (BI)*, which uses the values computed in the backward pass to choose the most profitable increase. Computational experiments have shown that BI minimizes the number of iterations. However, IBD keeps a significant part of the search space out of the legal range for many iterations. This reduces the number of state transitions to be calculated, resulting in more, but faster iterations. When it comes to the choice between BI and IBD, there is not one best choice. For both variants, instances can be constructed that show benefits for that order. However, it seems to be a safe choice to choose Best Increment, which minimizes the number of iterations.

Both variants of *Optimal Policy Iteration* were implemented using *Java*. For comparison, a simple Dynamic Programming approach as well as an Evolutionary Algorithm (EA) were implemented.

As predicted, *OPI* outperformed the simple DP algorithm when capacities were increased but the number of dimensions remained constant. This meets the expectations set by the improvements in worst case running time. The outcome between *OPI* and the EA was not that clear. For the EA, the number of dimension seems to have only linear impact. This makes it faster with more than 6 dimensions. However, the Evolutionary Algorithm is not guaranteed to yield optimal results.

An important negative result of this thesis is described in chapter 11. The possibility of using a slightly modified version of *OPI* for computing optimal policies when the prices are determined by a scenario tree was a main motivation of pursuing the algorithm's exploration. We showed, however, that introducing scenario trees means that, given an optimal policy for a problem instance, the current size the neighborhood is not sufficient for finding an optimal solution when creating a new instance by adding one unit.

Optimal Policy Iteration turned out to be a reliable method of constructing optimal solutions to the allocation problem considered in this thesis. It significantly improves the worst case running time of simple DP with only a moderate increase in implementation complexity.

References

[1] Alfred Kalliauer. Mathematische Entscheidungsmodelle aus der Praxis – fuer den Alltag, Tagung Schulmathematik, TU Wien, 2006.

[2] Alfred Kalliauer und Andrea Kalliauer. "Optimal Policy Iteration" in Dynamic Programming and Application to multi-dimensional Allocation Problems. In *International Conference OR2006*, Sep. 2006, Karlsruhe.

[3] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[4] Guenther R. Raidl Christoph Bonitz, Alfred Kalliauer. An incremental dynamic programming algorithm for multi-dimensional allocation problems. In *EURO XXII Conference, Prague*, 2007.

[5] Thomas H. Cormen, Charles E. Leierson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, second edition (paperback) edition, 2001.

[6] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. Algorithms. Draft.

[7] Reinhard Diestel. *Graph Theory*. Springer-Verlag, Heidelberg, third edition edition, 2005.

[8] Reinhard Diestel. *Graphentheorie*. Springer-Verlag, Heidelberg, 3. auflage edition, 2006.

[9] Patrick Jaillet, Ehud I. Ronn, and Stathis Tompaidis. Valuation of commodity-based swing options. *Management Science*, 50:909–921, 2004.

[10] Dieter Jungnickel. *Graphs, Networks and Algorithms (Algorithms and Computation in Mathematics) Für Kunden: Stellen Sie Ihre eigenen Bilder ein. Hier reinlesen und suchen Graphs, Networks and Algorithms (Algorithms and Computation in Mathematics)*. Springer, 2004.

[11] Alfred Kalliauer and Andrea Kalliauer. Computation model for option evaluation on probability trees using excel. In *Conference OR2002, Klagenfurt*, 2002.

[12] Alfred Kalliauer and Andrea Kalliauer. A variation of th calculation scheme for stochastic dynamic programming and its application to option pricing and allocation problems. In *21st IFIP-Conference on Systems Modeling and Optimization, Sophia-Antibes*, 2003.

[13] Kalliauer Alfred. Modellierung von kombinierten, mehrdimensionalen SWING-Optionen, Internationale Energiewirtschaftstagung, TU Wien, 2007.

[14] R.E. Larson. *State Increment Dynamic Programming*. Elsevier, 1968.

[15] Jun Morimoto, Garth Zeglin, and Chris Atkeson. Minimax differential dynamic programming: Application to a biped walking robot. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1927–1932, 2003.

[16] Richard S. Sutton. Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 353–357. Morgan Kaufmann, 1991.

APPENDIX

A Problem Instance File Format

CCCMAP-instances are stored in ASCII-encoded text-files with the UNIX-newline-character as line separator. Such a file has three sections: A first section describing problem dimensionality and constraints, a second section containing the actual profits from allocations and a third section containing the best known solution.

- The first line starts with an #-sign and is ignored. This is where a textual description of the problem can be placed.
- The second line contains an integer, which is the number of dimensions.

- The third line contains an integer for each dimension, signifying its capacity.
- The fourth line contains the number of time steps.
- The fifth line contains the global allocation limit per time step.
- The sixth line contains the time-specific allocation limit for each time step.
- The seventh line is left blank.
- For each time step there are as many lines as there are dimensions, followed by a single blank line. Each nonempty line represents a dimension and contains real values, which are the profits that can be realized by allocating units from this dimension for this time step, starting with the profit for the first allocation, ending with the profit for the n-th dimension, where n is the global allocation limit per time step. The values are separated by single space characters.
- The next set of lines contains the decision variables for the currently best-known solution to this problem instance. Each line represents one dimension and contains one integer for each time step.
- After one blank line, the last line contains the objective function value of the best known solution.

If there is no known legal solution yet, the decision variables and the objective function value are set to zero, indicating the trivial solution of not selling anything.

The following file shows a problem instance with two dimensions (each having capacity 5), 10 time steps, a global allocation limit of two units per time step, and all allocation limits specific to time steps set to two as well. No solution is known yet. As an example for objective value contribution, the first allocation in the second dimension on the third time step contributes 0.23471394.

```
# Automatically generated output
2
5 5
10
2
2 2 2 2 2 2 2 2 2 2

0.0781024 0.076438956
0.13969767 0.036287837

0.30170012 0.20553117
0.46896946 0.17205842

0.01167334 0.011118219
0.23471394 0.06047001

0.6273717 0.13196075
0.22822744 0.14278845
```

```
0.13320045 0.08295837
0.68440807 0.078431234

0.015616558 0.005626765
0.018131502 0.017843708

0.12342043 0.10180663
0.039027095 0.026482176

0.05179189 0.042044643
0.8425374 0.45524323

0.001603663 9.1767317E-4
0.32266328 0.16260414

0.08939438 0.056387104
0.04071935 0.034606844

0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0

0.0
```