Enhanced Policy Iteration Methods for Optimal Maintenance Scheduling

Konsta Parkkali

School of Science

Bachelor's thesis
Espoo 27.8.2021

Supervisor

Prof. Antti Punkka

Advisor

MSc Jussi Leppinen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the bachelor's thesis

Author Konsta Parkkali

Title Enhanced Policy Iteration Methods for Optimal Maintenance Scheduling

Degree programme Engineering Physics and Mathematics

Major Mathematics and Systems Sciences **Code of major** SCI3029

Teacher in charge Prof. Antti Punkka

Advisor MSc Jussi Leppinen

Date 27.8.2021 **Number of pages** 24 **Language** English

Abstract

Markov decision processes (MDPs) are widely used to help decision making in situations where future outcomes are partially random. The standard methods for solving the optimal policy of an MDP are Value Iteration and Policy Iteration. However, these methods become impractical in some cases. Value Iteration tends to converge very slowly in discounted problems with a discount factor close to 1, whereas the computation times with Policy Iteration depend strongly on the size of the state space.

In this thesis, we implement Modified Policy Iteration algorithm to a multi-component maintenance scheduling model based on a discounted MDP. The optimal maintenance schedule of this model was previously computed with Policy Iteration, which greatly restricted for example the number of components in the model. We further enhance the performance of Modified Policy Iteration with Gauss-Seidel method and Anderson Acceleration. We compare the different algorithms derived from these methods in terms of the computation times. We further analyze the computation times of the algorithms by varying both the discount factor and the size of the state space.

Modified Policy Iteration enabled us to solve the optimal maintenance schedule much faster than previously. The algorithm also required far less computational memory to operate when compared to Policy Iteration. This allowed us to solve also much larger problems in terms of the size of the state space. Both Gauss-Seidel method and Anderson Acceleration were successful in decreasing the computation time of Modified Policy Iteration. Anderson Acceleration was especially efficient when the discount factor was very close to 1. The best algorithm in terms of computation time was achieved by combining both Gauss-Seidel method and Anderson Acceleration.

Keywords Markov decision process, Policy Iteration, Modified Policy Iteration, Gauss-Seidel, Anderson Acceleration, maintenance scheduling

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Tekniikan kandidaatintyön tiivistelmä

Tekijä Konsta Parkkali

Työn nimi Tehostetut ohjauksen iterointi menetelmät optimaaliseen huollon aikatauluttamiseen

Koulutusohjelma Teknillinen fysiikka ja matematiikka

Pääaine Matematiikka ja systeemitieteet **Pääaineen koodi** SCI3029

Vastuuopettaja Prof. Antti Punkka

Työn ohjaaja DI Jussi Leppinen

Päivämäärä 27.8.2021 **Sivumäärä** 24 **Kieli** Englanti

Tiivistelmä

Markov-päätösprosessi on matemaattinen runko malleille, joita käytetään päätöksenteon apuna, kun tulevaisuuden tapahtumat ovat epävarmoja. Markov-päätösprosessin ratkaisu on prosessin optimaalinen ohjaus. Kaksi tavallisinta menetelmää Markov-päätösprosessin ratkaisemiseen ovat arvoiterointi (engl. Value Iteration) ja ohjauksen iterointi (engl. Policy Iteration). Nämä menetelmät ovat kuitenkin joskus epäkäytännöllisiä. Arvoiterointi konvergoituu erittäin hitaasti diskontatuissa ongelmissa, joissa diskonttauskerroin on lähellä lukua yksi. Ohjauksen iteroinnin laskenta-ajat taas riippuvat vahvasti tila-avaruuden koosta.

Tässä kandidaatintyössä implementoidaan muokattu ohjauksen iterointi algoritmi huollon aikataulutus malliin, jonka runkona on diskontattu Markov-päätösprosessi. Mallin ratkaisu kertoo parhaan tavan huoltaa monikomponenttinen systeemi, jonka komponenttien välillä on erilaisia riippuvuuksia. Mallin optimaalinen huoltoaikataulu ratkaistiin aikaisemmin ohjauksen iteroinnilla, mikä rajoitti esimerkiksi komponenttien mahdollista määrää. Muokattua ohjauksen iterointi algoritmia parannetaan edelleen Gauss-Seidel-metodilla ja Anderson-kiihdytyksellä (engl. Anderson Acceleration). Näistä metodeista johdettuja algoritmeja vertaillaan laskenta-ajan perusteella. Algoritmien laskenta-aikoja analysoidaan lisäksi vaihtelemalla diskonttauskerrointa ja tila-avaruuden kokoa.

Työ osoitti, että muokattu ohjauksen iterointi ratkaisee huollon aikataulutus mallin optimaalisen huoltoaikataulun paljon aikaisempaa lyhyemmässä ajassa. Algoritmi vaati myös paljon vähemmän laskennallista muistia toimiakseen verrattuna ohjauksen iterointiin. Tämä mahdollisti myös tila-avaruudeltaan paljon suurempien ongelmien ratkaisemisen. Sekä Gauss-Seidel-metodi että Anderson-kiihdytys pienensivät onnistuneesti muokatun ohjauksen iterointi algoritmin laskenta-aikaa. Anderson-kiihdytys oli erityisen tehokas silloin, kun diskonttauskerroin oli erittäin lähellä lukua yksi. Laskenta-ajan kannalta paras algoritmi saatiin yhdistämällä Gauss-Seidel-metodi sekä Anderson-kiihdytys.

Avainsanat Markov-päätösprosessi, ohjauksen iterointi, muokattu ohjauksen iterointi, Gauss-Seidel, Anderson-kiihdytys, huollon aikataulutus

Contents

1 Introduction **1**

2 Literature review **2**

3 Multi-component model review **4**

4 Comparison of selected methods **9**
4.1 Selected methods 9
4.2 Theory 9
4.3 Modified Policy Iteration 11
4.4 Gauss-Seidel method 13
4.5 Anderson Acceleration 15
4.6 Results 18

5 Conclusion **22**

1 Introduction

Modern systems often compose of many components. Without maintenance, these components deteriorate and become prone to breaking. If a critical component breaks, the whole system depending on it will fail. These failures can be extremely dangerous, for example the engine of an airplane breaking in the middle of a flight. System failures can also cause additional costs as the system cannot be in use before the broken component is fixed or replaced. On the other hand, maintaining components yields different costs, too. Hence, it is beneficial to use the components as long as possible while avoiding failures.

The randomness of the component breakdowns makes it hard to plan a good maintenance schedule. In addition, there can be dependencies between the components. For example, it can be cheaper to maintain the components of a system together than separately. Thus, mathematical modelling is often needed to get a good insight on how these systems should be maintained. One such option is Markov decision process (MDP; Howard, 1960), which is able to tackle the randomness and the complex structures of these systems. MDPs have been widely used in solving the optimal maintenance schedule of different maintenance problems, such as highway pavement maintenance (Puterman, 1994).

Leppinen et al. (2021) introduce a mathematical maintenance scheduling model for multi-component systems, with different component dependencies. Every component is critical and breaks according to some distinct probability distribution. Kokkonen (2021) extends this model by allowing components to have different quality types. For example, a 'bad' component is more likely to break at a certain age than a 'good' component. These quality types can be determined by inspections. The model is based on a discounted Markov decision process, and the optimal maintenance schedule is calculated with Policy Iteration. The downside of Policy Iteration is that it limits for example the number of components in the system, as adding more components quickly increases the computational complexity.

The optimal policies of Markov decision processes can also be calculated approximately. Puterman (1994) introduces an approximate version of Policy Iteration called Modified Policy Iteration, and a few variants that can improve its performance including the Gauss-Seidel method. Another, more recent method for enhancing the performance of Modified Policy Iteration is Anderson Acceleration (Geist and Scherrer, 2018). This thesis compares the applicability of these methods to the maintenance scheduling model presented by Leppinen et al. (2021) and Kokkonen (2021).

The rest of this thesis is organized as follows. Chapter 2 presents shortly the different methods used for solving MDPs. In Chapter 3 we introduce the maintenance scheduling model, for which we will apply the selected methods. In Chapter 4 we review the basic theory behind MDPs, present the algorithms for the selected methods and compare the performance of these methods in terms of computation time using varying parameters for the model. Chapter 5 concludes the results and discusses some possible future development ideas.

2 Literature review

Markov decision process is an extension of a discrete time Markov chain, where the state of the system can also be affected with actions. These actions have costs that can be discounted, meaning that future costs are seen less significant than immediate costs. There are also other variants of MPDs. For example, a state can yield a distinct reward when visited (Boutilier et al., 1995), or the state of the system might not be precisely known (Cassandra, 1998). The objective of MDPs is to find an optimal set of actions, referred to as the optimal policy, which for example minimizes the operating costs of some system. In this thesis, we consider discounted Markov decision processes where actions have costs.

The two main algorithms for computing the optimal policy of an MDP are Value Iteration and Policy Iteration. A third approach for solving MDPs is linear programming, although this has been considered to be impractical for most problems as the computational complexity is often very high (Littman et al., 2013).

Value Iteration (VI) is the most widely used algorithm for solving MDPs. VI produces an ε-optimal solution in the sense that with small enough values of ε, an optimal solution is obtained. VI is often used as it is easy to implement in practise. However, VI usually converges slowly with high discount factors. Puterman (1994) mentions several variants that decrease the computation time of VI. He refers to them as splitting methods, which include Gauss-Seidel and Jacobi Value Iterations, and over-relaxation. All these methods can also be combined with each other.

Shlakhter (2010) introduces a General Accelerated Value Iteration algorithm (GAVI). Two different operators can be used in the algorithm: a projective operator and a linear extension operator. Slakhter's thesis shows that in certain types of MDPs, GAVI can perform significantly better than VI. GAVI can also be combined with the splitting methods.

In more recent years, convex optimization methods have been used to create faster algorithms for solving MDPs (Grand-Clément, 2021). For example, Geist and Scherrer (2018) introduce Anderson Accelerated Value Iteration algorithm (AA-VI). Anderson Acceleration is a method for solving fixed-point iterations, and it originates from the 1960s. Nevertheless, it has not been implemented in dynamic programming problems since very recently. Although there is no proof of the convergence of AA-VI, experimental results have shown it to consistently outperform Value Iteration.

Other ideas from convex optimization used to improve VI's performance are Nesterov's acceleration and momentum (e.g. Goyal, 2019; Akian et al., 2020). Goyal (2019) shows that these methods can significantly improve the performance of VI. He also proves that these methods converge to the optimal policy under certain conditions. Akian et al. (2020) introduce a dth-order algorithm called dA-VI, which is also based on Nesterov's acceleration. In the paper, proofs of convergence were made with less restrictive conditions than in the proof of Goyal (2019).

The second main algorithm for solving MDPs, Policy Iteration (PI), is proved to converge to the optimal policy in a finite number of steps, and to halt when this is achieved (Puterman, 1994). At each iteration of PI, a linear system of equations is solved. With problems with large state spaces, the computational complexity of solving this system of equations becomes infeasibly large, which is why PI is often an impractical solution method. However, in small problems PI is usually the preferred method.

Puterman (1994) introduces a modified version of the Policy Iteration algorithm, called Modified Policy Iteration (MPI). This algorithm combines the ideas of Value Iteration and Policy Iteration to achieve improved performance especially in discounted MDPs. In the algorithm, the linear system of equations in Policy Iteration is solved approximately with a fixed-policy Value Iteration. Puterman (1994) concludes: *"Therefore in practice, value iteration should never be used. Implementation of modified policy iteration requires little additional programming effort yet attains superior convergence."*

Puterman (1994) also presents action elimination procedures, which can improve the performance of MPI even further. This method tries to find non-optimal actions and remove them from the problem. This reduces the number of possible policies, and consequently the computation time needed.

3 Multi-component model review

This chapter introduces the maintenance scheduling model developed by Leppinen et al. (2021) and extended by Kokkonen (2021). In this model, a system composes of $N = \{1, ..., |N|\}$ components. These components age, which makes them prone to breaking. The components can be maintained, which makes them as good as new. Every component in these systems is critical, which means that the breakdown of any single component will cause the whole system to fail as well. The breakdowns of the components are considered to be independent of each other.

The components can have different quality types. These quality types affect how the components break. For example, a component with a 'bad' quality type will break more likely at a certain age than a component with a 'good' quality type. Each quality type of every component breaks according to a distinct probability distribution. The quality types can be determined by inspections. Let $M \subseteq N$ denote the set of components that can be inspected.

The dependencies between the components are modelled with a directed graph $G = (V, D)$. Here, V is the set of nodes. The root node 0 is the starting point of every maintenance session. There are two different kinds of action nodes: maintenance action nodes n for every component in N and inspection nodes m for every component in M. Maintenance action nodes n refer to the action of maintaining the corresponding component. Similarly, inspection nodes m refer to the action of inspecting the component. D is the set of directed arcs (i, j), where i is the start node and j is the end node of the corresponding arc. The arcs have weights $c_{ij} \geq 0$, which represent the costs of action j, if action i is also completed.

If a node is connected to the root node, the corresponding action can be done on its own. A node that is not connected to the root node is representing a structural dependency: the action cannot be completed without completing another action as well. There can also be multiple arcs leading to one node. This is a representation of an economical dependency: the maintenance cost is dependent on the other maintained components. Every time at least one component is maintained or inspected, a fixed set-up cost $c_0 \geq 0$ is paid. A broken component is always maintained. Maintaining a broken component causes an additional cost, called a corrective replacement surplus cost. For component i, this cost is denoted by $r_i \geq 0$.

Figure 1 is the case example used by Leppinen et al. (2021). We will refer to this as system 1, because we use it as an example in the following chapter. Here, the system composes of four components. The components are E1, E2, C and W. In addition there is a dummy node 'DE12', which represents the decision to dismantle both E1 and E2. For example, in order to maintain component W, both E1 and E2 must be dismantled. In this system all components have only one quality type.

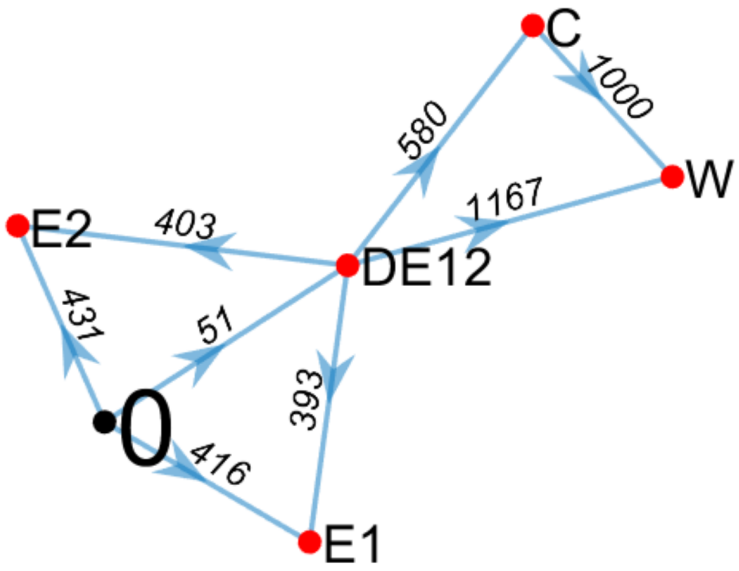

Figure 1: Case example used by Leppinen et al. (2021).

The model operates in discrete time. At each maintenance instance $t_{k+1} = t_k + \Delta t,\ k \in \mathbb{N}$ the components can be maintained or inspected. The time taken for these actions is considered to be negligible compared to the maintenance interval Δt. The ages of the components are known. We denote the ages of the components at maintenance instance t_k by $g_k = t_k - \tau \in \mathbb{R}^{|N|}$, where $\tau \in \mathbb{R}^{|N|}$ is the maintenance history of the components. Here, τ_i is the last maintenance instance when component i was maintained.

Between two consecutive maintenance instances, the components either break or age. The breakdown probabilities depend on the quality types of the components. For simplicity, let each component i have Q quality types, which are denoted by $q_i \in \{1, 2, ..., Q\}$. We model the probability of component i breaking before the age $(g_k)_i$ with a cumulative distribution function $\Phi_{q_i}((g_k)_i)$. If the quality of component i is not known, a joint distribution is used. This joint distribution is defined as

$$\Phi_i((g_k)_i) = \sum_{q_i=1}^{Q} \Phi_{q_i}((g_k)_i) P_{q_i}((g_k)_i),$$

where $P_{q_i}((g_k)_i)$ is the probability of component i being of quality type q_i at the age of g_k. These probabilities are also referred to as inspection probabilities, and can be calculated using the quality types' initial distributions.

It is assumed that only one component can break at a time. This assumption is not very strong, as Leppinen et al. (2021) state. With this assumption, we can calculate the conditional probabilities for the individual components breaking during

(t_k, t_{k+1}) and the probability for the whole system to keep operating during that same maintenance interval. These are denoted by $F_i(t_k)$ for component i and $R_{sys}(t_k)$ for the whole system. This probability $R_{sys}(t_k)$ is also considered as the reliability of the system. The system must be maintained so that $R_{sys}(t_k) \geq \rho$, where $\rho \in [0, 1)$ is the reliability threshold. For more details on these probabilities, see Leppinen et al. (2021) and Kokkonen (2021).

We can model the state of this system using the components' ages g_k, the inspection state I_k and the failure state f_k. The inspection state is a vector $I_k \in \{0, 1, ...\}^{|N|}$, where $(I_k)_i$ is the quality type of component i. If the quality type is unknown, we set $(I_k)_i = 0$. The failure state is a binary vector $f_k \in \{0, 1\}^{|N|}$, where $(f_k)_i = 1$ means that component i has broken. The possible age combinations g_k are limited by the reliability threshold constraint. From this point forward we consider only those age combinations g_k which satisfy this constraint. Now, the state of the system at t_k is denoted by a matrix s_k:

$$s_k := \begin{bmatrix} g_k^\top \\ I_k^\top \\ f_k^\top \end{bmatrix} \in \mathbb{R}^{3\times|N|}.$$

As a result of the reliability threshold constraint, the number of possible age combinations g_k is finite and can be denoted by h. Each component $i \in M$ has Q different quality types. For components that cannot be inspected, the quality type is always unknown. The number of possible inspection states is therefore $(Q + 1)^M$. However, not all inspection states are feasible with all age combinations. For example, a 'good' component can satisfy the reliability threshold at an older age than a 'bad' component. In addition, the quality type of a new component is always unknown as Kokkonen (2021) assumed. Between every two maintenance instances, one of the components $i \in N$ can break, or all components can remain functional. Thus, the number of possible outcome scenarios is $|N| + 1$ in total. Altogether, the number possible states $\mathbb{S}$ the system can be in is limited above by $h(Q + 1)^M(|N| + 1)$.

The action of maintaining and inspecting components at a maintenance instance t_k is denoted by $a_k = \{a_k^M \; a_k^I\}$, where $a_k^M \in \{0, 1\}^{|N|}$ is the decision of maintaining components and $a_k^I \in \{0, 1\}^{|M|}$ is the decision of inspecting components. If component i is maintained, we set $(a_k^M)_i = 1$. Similarly, we set $(a_k^I)_i = 1$ when choosing to inspect component i. In total, there are $2^N \times 2^M$ ways to choose a maintenance action for each state s_k. However, the number of feasible maintenance actions is smaller due to the reliability threshold constraint, the structural dependencies in the system, and since a broken component must be maintained. In addition, maintained components can not be inspected. We will denote the set of feasible actions at state s_k as $\mathbb{A}_{s_k}$. For the detailed formulation of these feasible actions, see Leppinen et al. (2021) and Kokkonen (2021).

When a feasible maintenance action $a_k \in \mathbb{A}_{s_k}$ at state s_k is chosen, the corresponding costs $C(s_k, a_k)$ can be determined from the graph G using Edmond's algorithm (Kleinberg and Tardos, 2006). This algorithm finds the cheapest possible way of executing the chosen maintenance action. If at least one component is maintained or inspected, the set-up cost c_0 is included in the cost. If component i has broken the surplus cost r_i is also included. These costs are discounted with a discount factor $\beta \in (0, 1)$.

After executing the chosen maintenance actions, the maintenance history τ is updated by

$$\tau_i = \max\{\tau_i, (a_k^M)_i t_k\}, \quad \forall i \in N.$$

The inspection states are also updated. If an inspected component i is maintained, the next inspection state $(I_{k+1})_i$ is zero as the quality type is no longer known. On the other hand, if a component is inspected, then the next inspection state $(I_{k+1})_i$ is set to match the quality type of the component q_i. After updating the maintenance history and inspection state, the system transitions to the next $|N| + 1$ states according to the transition probabilities determined before. These probabilities can be expressed as a row vector

$$P_{s_k}(a_k) = [F_1(t_k), \ \ldots \ , F_N(t_k)), R_{sys}(t_k)] \in \mathbb{R}^{1\times(|N|+1)}.$$

Using these probabilities together with the inspection probabilities, the probability to move from state s_i to s_j with action a_k can be calculated and is denoted by $P_{s_i s_j}(a_k)$.

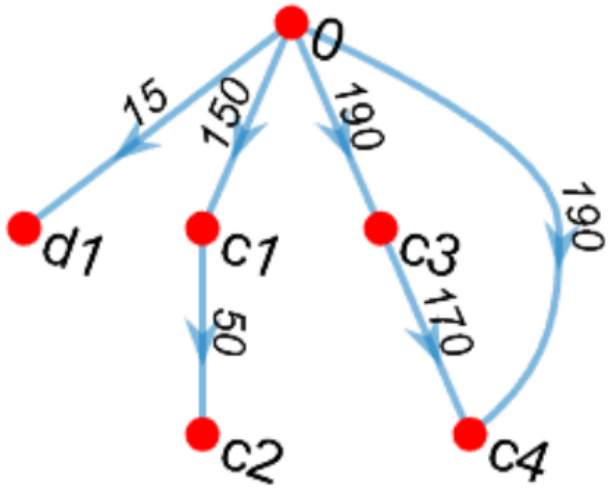

Figure 2: Case example used by Kokkonen (2021).

The system in Figure 2 is one of the case examples used by Kokkonen (2021). We will refer to it as system 2. Here c_1, c_2, c_3 and c_4 are the decisions to maintain

the corresponding four components and d_1 is the decision to inspect component 1. Component 1 has two different quality types: good and bad. The probability for the quality type of component 1 being good after maintenance is 0.75.

In both system 1 and 2, we set the reliability threshold to $\rho = 0.8$, the discount factor to $\beta = 0.99$, and use Weibull distributions as the cumulative distribution functions that model component breakdowns. In the following chapter, we use the maintenance interval Δt as a free parameter to adjust the size of the state space $\mathbb{S}$ of the systems.

4 Comparison of selected methods

4.1 Selected methods

The methods we chose for comparison are Modified Policy Iteration (MPI), Gauss-Seidel Modified Policy Iteration (GS-MPI), Anderson Accelerated Modified Policy Iteration (AA-MPI) and the combination of all of these, Anderson Accelerated Gauss-Seidel Modified Policy Iteration (AA-GS-MPI). Other methods were left aside as they were not compatible with our framework or did not improve the computation times of our numerical examples.

Jacobi Value Iteration differs from regular VI only if the probabilities P_{ii} of moving from some state back to itself are positive. This means that the system can be in the same state on multiple consecutive steps. In our model, this would require that all components are new, and all components are maintained on the very next maintenance instance. As this is not a very sensible action, the performance of Jacobi VI would be practically the same as VI.

Shlakhter (2010) mentions that General Accelerated VI is *"expected to be inefficient"*, with MDPs where the proportion of non-zero elements in transition matrices is less than 5%. In addition, for very sparse transition matrices (less than 0.1% non-zero elements) the performance of GAVI is considered to be poor. Since the transition matrices of the MDPs considered in this thesis are always very sparse, GAVI was discarded as an infeasible method.

The methods derived from Nesterov's acceleration (Goyal, 2019) can also be applied to the fixed-policy value iteration phase of MPI. However, the methods were only proved to converge in the case that the Markov chain produced by the policy is irreducible and reversible. In our model, not all policies produce an irreducible Markov chain. Leppinen et al. (2021) show that with the optimal policy, less than half of the states are reachable in some cases. Nevertheless, these methods were found to diverge in our numerical examples.

From the remaining methods, over-relaxation and action elimination procedures (Puterman, 1994) were not able to improve the performance of Modified Policy Iteration in our examples. Action elimination procedures were not able to find any non-optimal actions at all.

4.2 Theory

Before presenting the algorithms, we review the basic theory of MDPs. A discounted Markov Decision Process can be defined as a tuple $(\mathbb{S}, \mathbb{A}_s, \mathbf{P}, \mathbf{C}, \beta)$. In this thesis, $\mathbb{S}$ is a finite set of states and $\mathbb{A}_s$ is the finite set of actions that are feasible in state s. A stationary policy is a function $\pi : \mathbb{S} \to \bigcup_{s \in \mathbb{S}} \mathbb{A}_s$, that attaches each state s to

some feasible action $a \in \mathbb{A}_s$. $\mathbf{P} \in \mathbb{R}^{|\mathbb{S}|\times|\mathbb{A}|\times|\mathbb{S}|}$ is a kernel that models the transition probabilities from each state-action pair to the new states. The transition probabilities from state s to state s' with action a are denoted by $P_{ss'}(a)$. $\mathbf{C} \in \mathbb{R}^{|\mathbb{S}|\times|\mathbb{A}|}$ denotes the costs relating to each state-action pair, which are discounted with a discount factor $\beta \in (0, 1)$.

The long term running costs of a discounted MDP with a chosen policy π, are represented by a value vector $v^\pi \in \mathbb{R}^{|\mathbb{S}|}$. In this vector, $v^\pi(s)$ stands for the long term expected cost of the process when it starts at state s. This vector can be defined componentwise as

$$v^\pi(s) = E\left[\sum_{t=0}^{\infty} \beta^t C(s_t, \pi(s_t)) \mid s_0 = s\right], \qquad \forall s \in \mathbb{S}.$$

The dynamic programming principle states that for a fixed policy π, the value vector v^π is a solution to the Bellman equation:

$$v^\pi(s) = C(s, \pi(s)) + \beta \sum_{s' \in \mathbb{S}} P_{ss'}(\pi(s)) v^\pi(s'), \qquad \forall s \in \mathbb{S}. \tag{1}$$

The long term running costs v^π of a certain policy π can be solved analytically from the above system of linear equations (1) in matrix form by $v = (I - \beta P)^{-1} C$. The vector v^π can also be solved with an iterative algorithm called value computation (Puterman, 1994). Value computation is also referred to as fixed-policy value iteration or policy evaluation. Value computation is defined as $v_{n+1} = T_\pi v_n$, where

$$v_{n+1}(s) = C(s, \pi(s)) + \beta \sum_{s' \in \mathbb{S}} P_{ss'}(\pi(s)) v_n(s'), \qquad \forall s \in \mathbb{S}.$$

This iteration is proved to converge to v^π, with any initial guess $v_0 \in \mathbb{R}^{|\mathbb{S}|}$ (Puterman, 1994).

Let π^* be the optimal policy, which minimizes the expected long term costs. Then the value vector v^{π^*} is the solution to the Bellman optimality equation:

$$v^{\pi^*}(s) = \min_{\pi(s) \in \mathbb{A}_s} \left\{ C(s, \pi(s)) + \beta \sum_{s' \in \mathbb{S}} P_{ss'}(\pi(s)) v^{\pi^*}(s') \right\}, \qquad \forall s \in \mathbb{S}.$$

This value vector can be solved iteratively with Value Iteration. It is defined as an iteration $v_{n+1} = T v_n$, where

$$v_{n+1}(s) = \min_{\pi(s) \in \mathbb{A}_s} \left\{ C(s, \pi(s)) + \beta \sum_{s' \in \mathbb{S}} P_{ss'}(\pi(s)) v_n(s') \right\}, \qquad \forall s \in \mathbb{S}. \tag{2}$$

This iteration is proved to converge to the optimal value vector v^{π^*}, with any initial guess $v_0 \in \mathbb{R}^{|\mathbb{S}|}$ (Puterman, 1994).

Next we present the algorithm for Policy Iteration, which will form the base for our methods comparison.

Algorithm 1: Policy Iteration

initialization Select a policy π_0
for $n \geq 0$ **do**
  Value-determination
  Solve v^{π_n} from the system of equations (1)
  Policy-improvement
  Set $\pi_{n+1}(s)$ to the argument $\pi(s)$ which minimizes the right hand side of equation 2 for all $s \in \mathbb{S}$.
  if $\pi_{n+1} = \pi_n$ **then**
    return π_n
  end
end

When the Policy Iteration algorithm stops, the policy π_n will be equal to the optimal policy π^* with any initial policy π_0 (Puterman, 1994). The initial policy π_0 used in our implementation consists of the cheapest possible feasible actions in each state.

We solve the system of equations (1) approximately using the conjugate gradients squared method built in Matlab (function cgs). We set the tolerance to 10^{-6} and maximum number of iterations to 500.

4.3 Modified Policy Iteration

Next, we introduce the Modified Policy Iteration algorithm. In this algorithm, instead of solving the value vector v^{π_n} analytically, only a small number of value computation iterations is done to improve the estimate of this vector.

Algorithm 2: Modified Policy Iteration

initialization Select a value vector v_0, specify $\varepsilon > 0$, choose $M \in \mathbb{N}$
for $n \geq 0$ **do**
  Policy-improvement
  Set $\pi_{n+1}(s)$ to the argument $\pi(s)$ which minimizes the right hand side of equation 2 for all $s \in \mathbb{S}$.
  Set $u = Tv_n$
  if $|u(s) - v_n(s)| < \varepsilon(1-\beta)/2\beta \quad \forall s \in \mathbb{S}$ **then**
    return π_{n+1}
  end
  Partial value computation
  Set $v_{n+1} = (T_{\pi_{n+1}})^M u$
end

In this algorithm, the parameter M denotes the number of value computation iterations done in the partial value computation phase at each iteration of the algorithm. For example, $(T_\pi)^M$ means that the operator T_π is applied M times. This number M could also change from one iteration to another, or be dynamically set. However, in our numerical examples we were not able to find any changing or dynamic M that would outperform a fixed M. Notice that when $M \to \infty$, this algorithm essentially becomes Policy Iteration. In addition, if $M = 0$ the algorithm is the same as Value Iteration.

The parameter ε can be seen as the required accuracy of the algorithm. When MPI stops, the returned policy π_{n+1} will be ε-optimal in the sense that

$$|v^{\pi_{n+1}}(s) - v^{\pi^*}(s)| < \varepsilon/2 \quad \forall s \in \mathbb{S}. \tag{3}$$

Intuitively this means the following. We choose $\varepsilon = 1$ for example. Then the long term costs related to the returned policy π_{n+1} are at most 0.5 more than the costs related to an optimal policy. We use $\varepsilon = 1$ in the following examples, and we will return later to this choice of ε.

In our implementation, we use the cost vector $C(s, \pi_0(s))\ \forall s \in \mathbb{S}$ as our initial guess for v_0. Here π_0 is the same policy as the initial policy used in our implementation of PI. This initial guess v_0 is also used for other algorithms presented in this thesis.

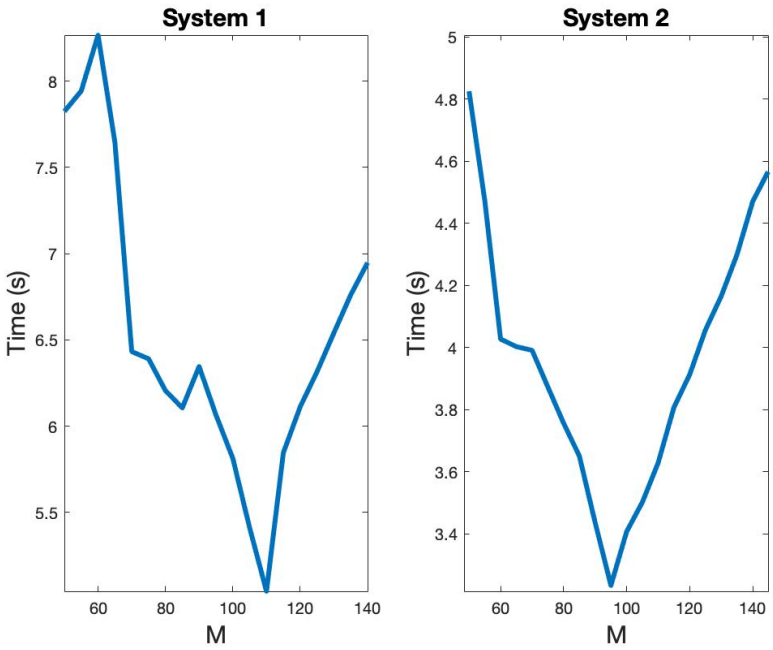

Figure 3: Comparing the choice of M in terms of computation time in systems 1 and 2.

In Figure 3 we compare the total running time of MPI with different values for the parameter M in systems 1 and 2. For M, we use the values from 50 to 140,

with a stepsize of 5. Here we have set $\Delta t = 0.7$ for system 1 and $\Delta t = 0.9$ for system 2. With low values of M, the approximation v_{n+1} is too poor for the next policy-improvement phase. This results in a large number of total iterations for the algorithm, and in a long total computation time. Too large values of M will only result in doing useless computations, as the approximation v_{n+1} is already good enough for the policy-improvement phase. The graphs are not smooth, as increasing M does not always decrease the number of total iterations needed. This makes it hard to choose the best M. In addition, the best choice of M depends on the parameters of the problem. According to Figure 3, a good choice of M seems to be around 100 with these settings.

4.4 Gauss-Seidel method

The Gauss-Seidel Modified Policy Iteration algorithm is otherwise similar to MPI, except instead of operators T and T_π it uses operators T^{GS} and T_π^{GS}, respectively. The Gauss-Seidel value computation can be defined as $v_{n+1} = T_\pi^{GS} v_n$, where

$$v_{n+1}(s) = C(s, \pi(s)) + \beta \sum_{s'<s} P_{s,s'}(\pi(s)) v_{n+1}(s') + \beta \sum_{s' \geq s} P_{s,s'}(\pi(s)) v_n(s'), \quad \forall s \in \mathbb{S}.$$

Similarly, the Gauss-Seidel Value Iteration can be defined as $v_{n+1} = T^{GS} v_n$, where

$$v_{n+1}(s) = \min_{\pi(s) \in \mathbb{A}_s} \left\{ C(s, \pi(s)) + \beta \sum_{s'<s} P_{s,s'}(\pi(s)) v_{n+1}(s') + \beta \sum_{s' \geq s} P_{s,s'}(\pi(s)) v_n(s') \right\}, \quad \forall s \in \mathbb{S}.$$

Here, the state space $\mathbb{S}$ can be seen as an indexed list, where $s' < s$ is true for every s' that has an index smaller than s. The updates are done in order, from the state with the smallest index to the state with the largest. It is worth noting that the order of the states s can significantly affect the performance of the Gauss-Seidel variations. The order of states used in our implementation is illustrated in Table 1 with a system of 2 components, where each component can be a maximum of 2 units old. We set *broken component* = 3 when no components have broken. If component inspections are possible, the inspection states are added before the last column *broken component*. The inspection states would start at the bottom with a zero and go up in order. This order of states was first chosen because it is easy to implement in practise. However, we noticed that this order is also very efficient in terms of computation time. For example, a reversed order performed much worse. Wingate et al. (2005) present an algorithm which reorders the states more efficiently. This order turned out to be very similar to the one illustrated in Table 1. As a result to the initial state order being so good, any small possible benefits of the new order were lost to the computational needs of this algorithm.

Table 1: Example of state ordering in GS-MPI.

state index	age of component 1	age of component 2	broken component
1	2	2	1
2	2	2	2
3	2	2	3
4	2	1	1
5	2	1	2
6	2	1	3
7	1	2	1
8	1	2	2
9	1	2	3
10	1	1	1
11	1	1	2
12	1	1	3

An additional requirement is needed to ensure that GS-MPI converges. For the initial value vector v_0, $Tv_0(s) \geq v_0(s)$ needs to hold for all $s \in \mathbb{S}$. This holds with the initial value vector chosen before.

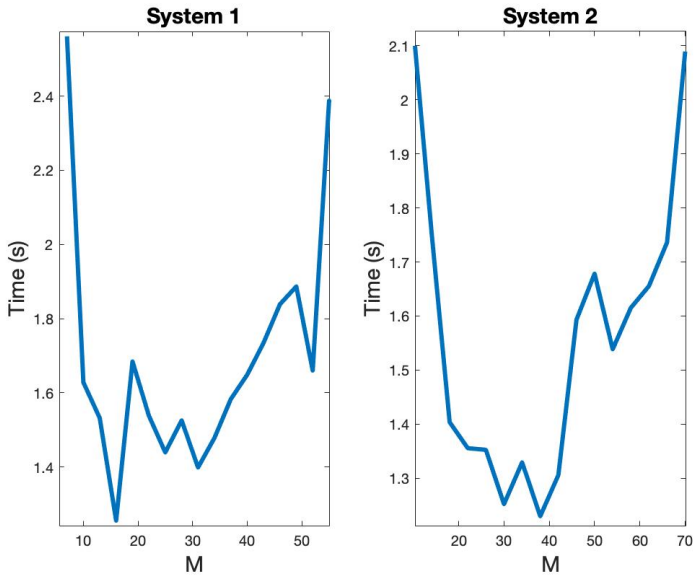

Figure 4: Comparing the choice of M in terms of computation time using GS-MPI.

In Figure 4 we compare the performance of GS-MPI with different values for M with similar settings as in Figure 3. Here, the best choice of M is around 20-40 considering both systems. This is far less than in MPI. This shows that the operator T_π^{GS} is more effective in estimating the value vector v^π than the operator T_π.

4.5 Anderson Acceleration

Next, we introduce Anderson Acceleration presented by Geist and Scherrer (2018). Let π be our chosen policy and T_π the corresponding operator. First, we need to calculate the current estimate v_m and memorize the last k estimates $v_{m-1}, ..., v_{m-k}$ for our value vector v^π. Here, k can be seen as the size of the memory. Then, a vector $\alpha \in \mathbb{R}^{k+1}$ is defined by

$$\alpha = \arg\min_{\alpha \in \mathbb{R}^{k+1}} \left\| \sum_{i=0}^{k} \alpha_i (T_\pi v_{m-k+i} - v_{m-k+i}) \right\| \text{ st. } \sum_{i=0}^{k} \alpha_i = 1. \tag{4}$$

Now, the new estimate v_{m+1} can be calculated by

$$v_{m+1} = \sum_{i=0}^{k} \alpha_i T_\pi v_{m-k+i}.$$

In equation (4) different norms can be used. Here, we use the L_2 norm since the solution to the optimization problem (4) can be solved analytically using the Karush-Kuhn-Tucker conditions. Here, we define $B_i = Tv_i - v_i \in \mathbb{R}^{|\mathbb{S}|}$ and $B = [B_{m-k}\ B_{m-k+1}\ ...\ B_{m-1}\ B_m] \in \mathbb{R}^{|\mathbb{S}| \times (k+1)}$. In addition, $\mathbf{1} \in \mathbb{R}^{k+1}$ refers to a vector of ones. According to Geist and Scherrer (2018), we can now calculate the vector α by

$$\alpha = \frac{(B^\top B)^{-1}\mathbf{1}}{\mathbf{1}^\top (B^\top B)^{-1}\mathbf{1}} \tag{5}$$

Next, we present the algorithm for Anderson Accelerated Modified Policy Iteration. In the algorithm, M is again the number of value computation iterations. We use Anderson Acceleration to accelerate the value computation phase. Here, we denote k_{max} as the maximum possible size of memory.

Algorithm 3: Anderson Accelerated MPI

initialization Select v_0, specify $\varepsilon > 0$, choose $M, k_{max} \in \mathbb{N}$
for $n \geq 0$ **do**
  Policy-improvement
  Set $\pi_{n+1}(s)$ to the argument $\pi(s)$ which minimizes the right hand side of equation 2 for all $s \in \mathbb{S}$.
  Set $u_0 = Tv_n$
  if $|u_0(s) - v_n(s)| < \varepsilon(1-\beta)/2\beta \quad \forall s \in \mathbb{S}$ **then**
    return π_{n+1}
  end
  Anderson Accelerated value computation
  for m ***from*** *0* ***to*** M **do**
    choose k_m such that $k_m \leq \min(m, k_{max})$
    $B_m = T_{\pi_{n+1}} u_m - u_m$
    Set $B = [B_{m-k_m} \; B_{m-k_m+1} \; ... \; B_{m-1} \; B_m]$
    Calculate α by (5)
    Set $u_{m+1} = \sum_{i=0}^{k_m} \alpha_i T_{\pi_{n+1}} u_{m-k_m+i}$
  end
  Set $v_{n+1} = u_{M+1}$
end

In the algorithm, k_m is the size of the memory at iteration m of the value computation phase. Note that if $k_m = 0$, the Anderson Accelerated value computation step reduces to a normal value computation step as the optimization problem (5) becomes trivial. Geist and Scherrer (2018) proposes that this is chosen by $k_m = \min(k_{max}, m)$. This would mean that the acceleration would be done on every iteration using the largest possible memory. However, we noticed that doing less accelerations results in better performance in terms of computation time. We suggest the following way to choose k_m:

$$k_m = \begin{cases} \min(k_{max}, M-5), & \text{if } m \geq M-5 \\ 0, & \text{otherwise.} \end{cases}$$

This choice of k_m means that we do $M - (M-5) + 1 = 6$ acceleration steps with a fixed memory in the end of the value computation phase. For example, let $M = 50$. Now, the acceleration is done only when $m \in \{45, 46, 47, 48, 49, 50\}$.

Figure 5 compares the time spent for AA-MPI as a function of the parameter M. Here we have again a similar setting than in Figures 3 and 4. We use a maximum memory size of $k_{max} = 20$. In our experience, too small a memory causes the algorithm to become inefficient as there is not enough information to do the acceleration. On the other hand, too large a memory will cause the algorithm to use very old value vector approximations in the acceleration, which can also decrease the performance. With very large memories, the matrix B can become badly scaled. This means that the scale of the elements in the matrix are so different that numerical accuracy is lost. In addition, with large values for the parameter M, the columns B_m of the

matrix become essentially zero vectors. This means that the matrix becomes close to singular. Both of these numerical issues can cause the algorithm to diverge. However, with good choices of M and k_m the algorithm seems robust.

Note that choosing the best M can be a little tricky. For example in system 1 the optimal choice of M seems to be around 25. However, when decreasing the size of M, the performance of the algorithm decreases rapidly. Thus, to ensure a good performance, slightly larger M is often the better choice.

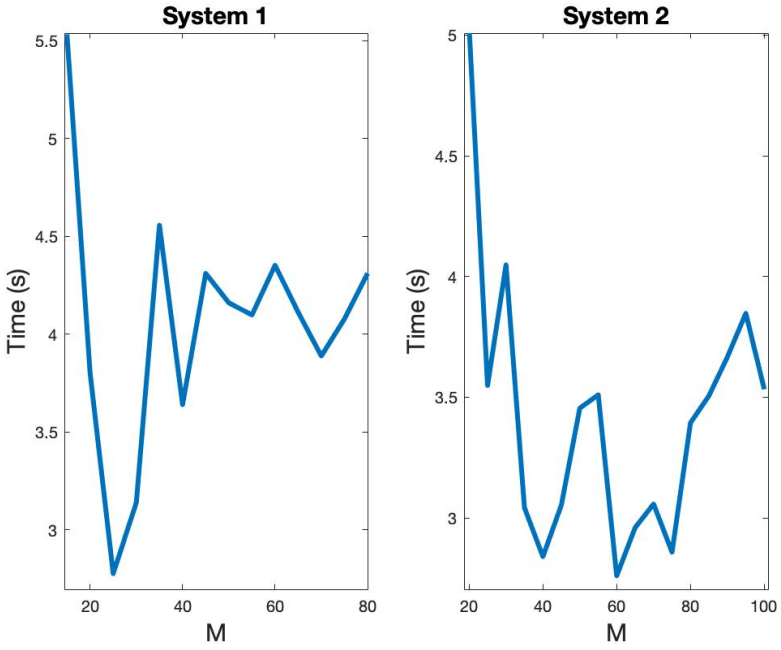

Figure 5: Comparing the choice of M in terms of computation time using AA-MPI.

Next, we will combine Anderson Acceleration with the Gauss-Seidel method. This is done by changing the operators T_π and T in algorithm 3 with the operators T_π^{GS} and T^{GS}, respectively. We also change the way k_m is chosen. Now we do only one acceleration on the very last step of the partial value computation phase. In our numerical examples, we found this to be the best option. This is achieved by choosing k_m by

$$k_m = \begin{cases} \min(k_{max}, M), & \text{if } m = M \\ 0, & \text{otherwise.} \end{cases}$$

We call this algorithm Anderson Accelerated Gauss-Seidel Modified Policy Iteration (AA-GS-MPI). In Figure 6 we compare the performance of AA-GS-MPI with different values of M with similar settings as in Figure 3. We have set the maximum memory again to $k_{max} = 20$. Here, the best choice of M is around 7-15.

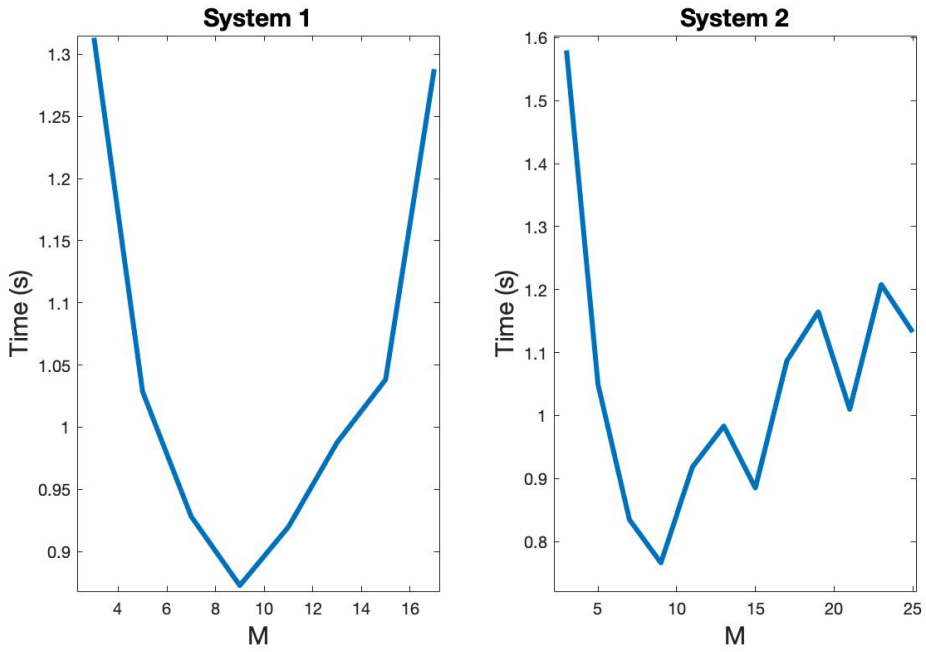

Figure 6: Comparing the choice of M in terms of computation time using AA-GS-MPI.

4.6 Results

In Figure 7 we compare the accuracies of the methods using system 1. We have set $\Delta t = 1.65$ to ensure that the methods run in approximately similar times. We have chosen the parameters $M = \{100,\ 30,\ 35,\ 8\}$ for MPI, GS-MPI, AA-MPI and AA-GS-MPI, accordingly. At every iteration of the algorithm, the current time and the accuracy of the value vector is measured and the progress is plotted. With the accuracy of a value vector, we mean the smallest ε for which equation 3 can be shown to hold. This figure is here to illustrate the fact that these methods based on Modified Policy Iteration can be as accurate as Policy Iteration when a small enough accuracy requirement ε is chosen. In theory, PI should reach an accuracy of $\varepsilon = 0$ when terminating. However, since the system of equations is solved numerically using the conjugate gradients squared method, $\varepsilon = 0$ is not quite reached. With our implementation of PI, the algorithm stops at around $\varepsilon = 1$. This is why we set the accuracy requirement to $\varepsilon = 1$ for the other algorithms as well. Notice that the Anderson Accelerated versions of MPI converge suddenly to very precise solutions. Thus, even if the accuracy is set to $\varepsilon = 1$, the result may be even closer to the optimum.

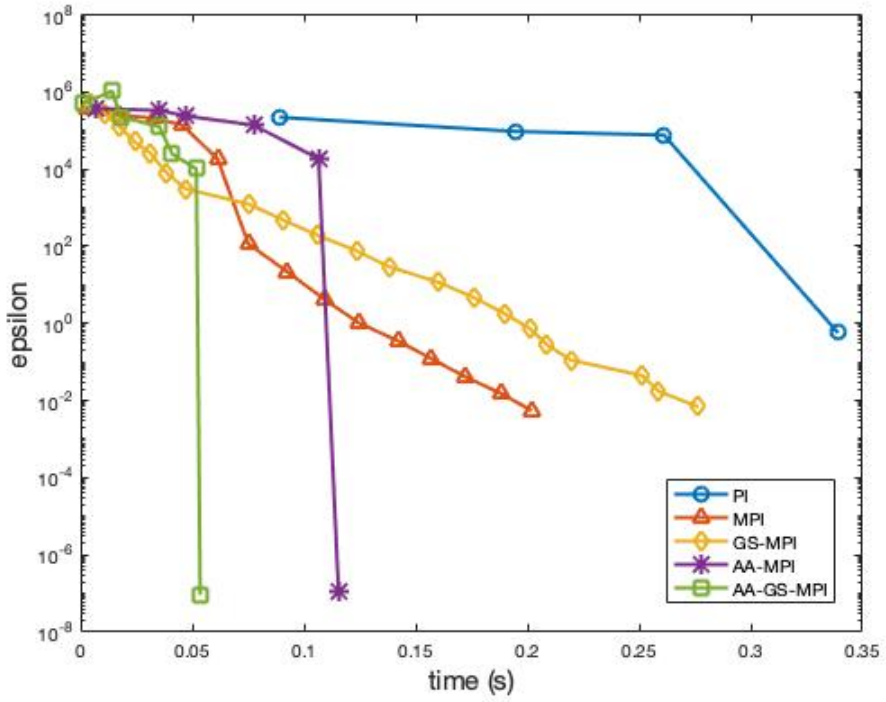

Figure 7: Comparison of the accuracies of the algorithms.

Next, we compare the total computation times of the algorithms with different sizes for the state space. The algorithms were run on version R2020b of MATLAB. The hardware used was a Macbook Pro (Mid 2014) with 2.6Ghz Intel Core i5 processor and with 8GB of RAM. We use the same parameters M as previously. We use system 1 and we adjust the size by changing the maintenance intervals Δt according to Table 2.

Table 2: Maintenance intervals used in Figure 8.

Maintenance interval Δt	0.7	0.6	0.5	0.4	0.3
Size of state space $\lvert\mathbb{S}\rvert$	95940	212055	529930	1634360	6972365

We illustrate the computation times in Figure 8. The black vertical lines represent the maintenance intervals. We can see that the computation time is approximately linear in the size of the state space. For reference, it took 184 seconds for Policy Iteration to finish with $\Delta t = 1.0$ and $|\mathbb{S}| = 16155$. This shows how slow it is to solve the system of linear equations (1), even when done with approximate methods.

In our numerical experiments, we found that the maintenance interval does not affect how the parameter M should be chosen. This makes it possible to choose a good M with a problem with a small state space, before moving on to solve bigger problems.

From Figure 8 we can see that the Gauss-Seidel variants of the algorithms perform

significantly better than algorithms without Gauss-Seidel method. The computation time of GS-MPI is about 85% smaller than the computation time of MPI. Anderson Acceleration is also able to decrease the computation times. For example, the computation times of AA-GS-MPI are about 15-45% less than the ones of GS-MPI.

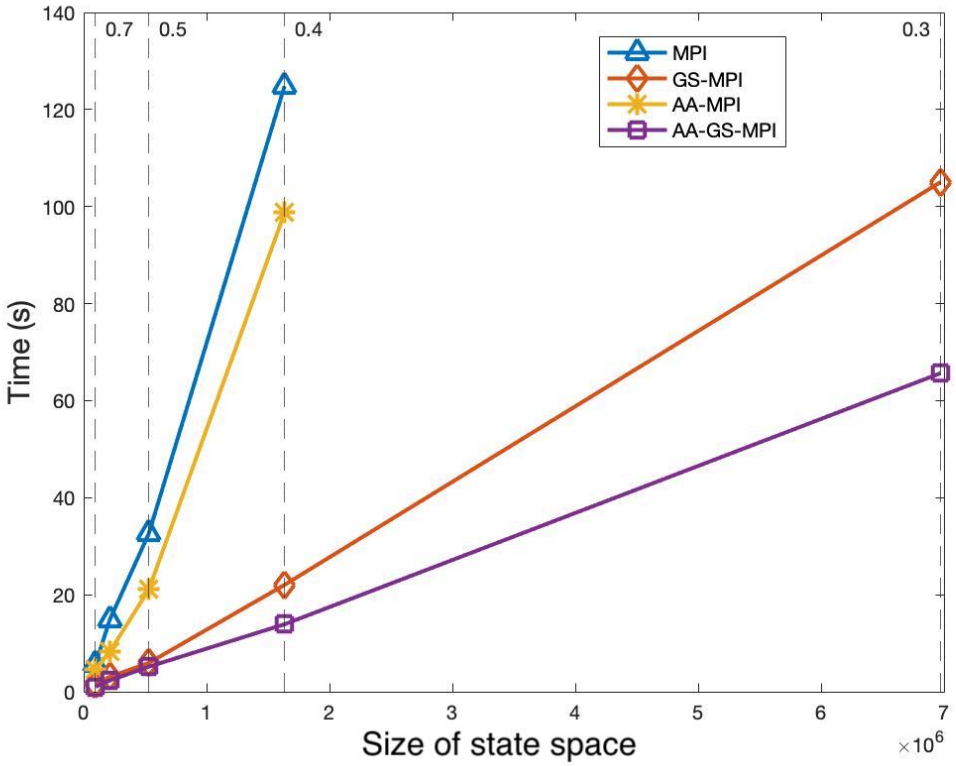

Figure 8: Algorithm running time comparison as a function of the size of the state space.

Next, we compare the performances with an increased discount factor. We change the discount factor β from 0.99 to 0.999 and use the maintenance intervals in Table 3. This change in discount factor slows the convergence of the value computation phase significantly. This means that a larger value for the parameter M could be more efficient. Here, we will use $M = \{200,\ 80,\ 50,\ 8\}$ for MPI, GS-MPI, AA-MPI and AA-GS-MPI, respectively.

Table 3: Maintenance intervals used in Figure 9.

Maintenance interval Δt	0.9	0.8	0.7	0.6	0.5
Size of state space $\lvert\mathbb{S}\rvert$	27330	49225	95940	212055	529930

The results are in Figure 9. We can see that the increase in discount factor decreases the performance of MPI and GS-MPI significantly. With $\Delta t = 0.7$, MPI is

approximately 15 times slower when the discount factor was increased from $\beta = 0.99$ to $\beta = 0.999$. Similarly, GS-MPI is over 6 times slower. On the other hand, the Anderson Accelerated versions are not affected much. With $\Delta t = 0.5$, AA-GS-MPI became only 23 % slower when the discount factor was increased. This shows that Anderson Acceleration is especially efficient with problems with high discount factors, as normal value computation converges very slowly.

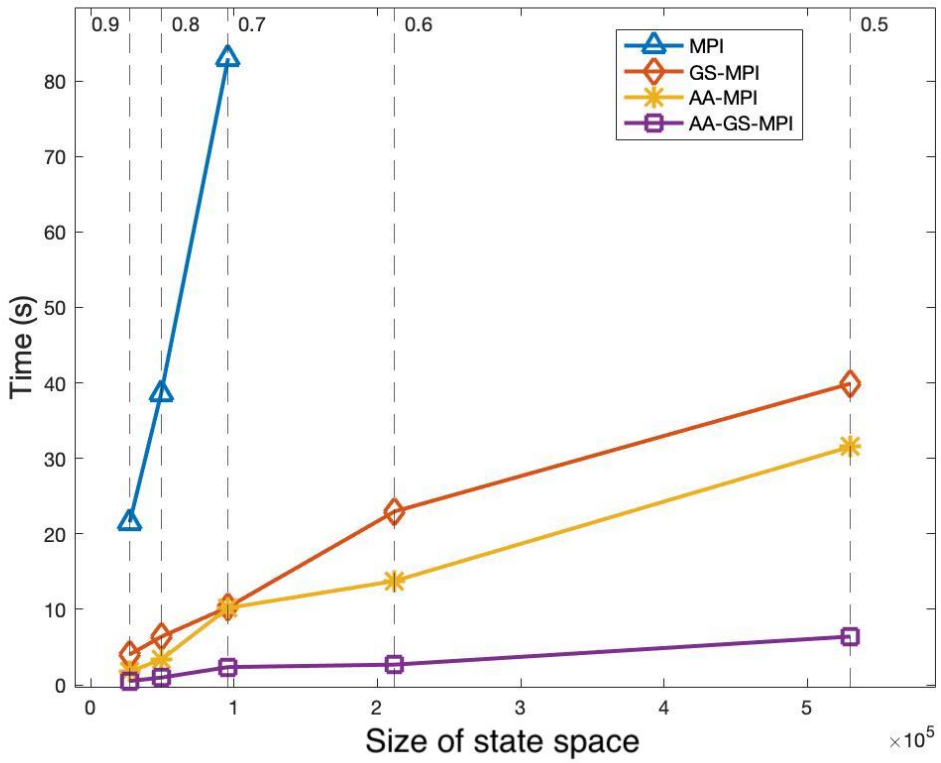

Figure 9: Comparing the computation times with a discount factor $\beta = 0.999$.

The best algorithm it terms of computation time turned out to be the combination of Gauss-Seidel method and Andersson Acceleration. However, 8GB of RAM became quickly the limiting factor with the calculations. For testing our algorithm with even larger problems, we switched on a windows 10 desktop computer with Intel Xeon Platinum 8176 CPU and 32GB of RAM. The version of MATLAB was R2021a. With AA-GS-MPI, we were able to solve the optimal maintenance schedule of system 1 with $\Delta t = 0.18$ and $\beta = 0.99$. The size of the state space with this choice of maintenance interval was 91 685 270. This calculation took 29 minutes and 46 seconds. With shorter maintenance intervals than $\Delta t = 0.18$, we run out of memory when setting up the problem.

5 Conclusion

This thesis improved the solution method used for solving the optimal maintenance policy of a maintenance scheduling model developed by Leppinen et al. (2021) and extended by Kokkonen (2021). In this model, a system consists of many components. These components can have different quality types, which affect how the components break when aging. The quality types can be determined by inspections. A maintenance action consists of the decision to maintain or inspect the components. The state of the system is defined with the ages and quality types of the components in addition with the possibility that some component has broken. This model can be formulated as a discounted Markov decision process. Previously the optimal maintenance policy was solved with Policy Iteration. The downside of Policy Iteration is that the computational complexity grows quickly when the size of the state space increases. This limits for example the number of components and the length of the maintenance interval.

First, we implemented Modified Policy Iteration to the maintenance scheduling model. Then we enhanced its performance even further with Gauss-Seidel method and Anderson Acceleration. With Modified Policy Iteration, we were already able to solve much larger problems in reasonable times than with Policy Iteration. This was expected as Policy Iteration is known to be impractical for solving MDPs with large state spaces. Gauss-Seidel method proved to improve the performance of MPI greatly. This is partly due to the way in which we ordered the states. Anderson Acceleration also improved the performance of MPI, especially in problems with discount factors very close to 1. The best algorithm turned out to be the combination of both of these methods: Anderson Accelerated Gauss-Seidel Modified Policy Iteration. Depending on the parameters of the problem, AA-GS-MPI was up to 97% faster than MPI. Although these solution methods based on Modified Policy Iteration are approximate, we were able to obtain just as accurate solutions as with Policy Iteration.

The challenge with these algorithms is that the parameters of the model, such as the discount factor, affect how the parameters of the algorithms should be chosen. When solving a new problem, a good choice of parameters should be first chosen by testing the algorithms with a reduced state space. This can be done by increasing the maintenance interval. Fortunately, with Anderson Accelerated Gauss-Seidel Modified Policy Iteration, the choice of good parameters seemed to be roughly constant.

The limiting factor of this model and solution method turned out to be memory usage. The explicit enumeration of all of the states takes up quickly the RAM of a standard laptop or desktop computer. Using a computer with 32 GB of RAM, we were able to solve a problem with 91 million states in just under 30 minutes. This is three orders of magnitude larger than the state space of the largest problem we were able to solve with Policy Iteration. However, with more efficient memory usage the number of states could be increased even further.

The computation times of the algorithms used in this thesis could be reduced with

parallelization. The steps in Modified Policy Iteration can largely be done in parallel using multiple processors. When done synchronously, this would not change any theoretical aspects of the algorithm. With the Gauss-Seidel variants, small changes would be needed (see e.g. Shang, 2009). Bertsekas and Yu (2010) propose a more efficient way of implementing this parallelization to Modified Policy Iteration. In their implementation the processors operate asynchronously. However, some additional requirements are imposed to ensure the convergence for this approach.

References

M. Akian, S. Gaubert, Z. Qu, and O. Saadi. Multiply accelerated value iteration for non-symmetric affine fixed point problems and application to markov decision processes. *arXiv preprint arXiv:2009.10427*, 2020.

D.P. Bertsekas and H. Yu. Distributed asynchronous policy iteration in dynamic programming. In *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1368–1375. IEEE, 2010.

C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *IJCAI*, volume 14, pages 1104–1113, 1995.

A.R. Cassandra. A survey of pomdp applications. In *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*, volume 1724, 1998.

M. Geist and B. Scherrer. Anderson acceleration for reinforcement learning. In *EWRL 2018 - 4th European workshop on Reinforcement Learning*, France, 2018.

J. Goyal, V. Grand-Clément. A first-order approach to accelerated value iteration. *arXiv preprint arXiv:1905.09963*, 2019.

J. Grand-Clément. From convex optimization to mdps: A review of first-order, second-order and quasi-newton methods for mdps. *arXiv preprint arXiv:2104.10677*, 2021.

R.A. Howard. *Dynamic programming and markov processes.* John Wiley, 1960.

J. Kleinberg and E. Tardos. *Algorithm design.* Pearson Education India, 2006.

S. Kokkonen. A condition-based maintenance scheduling model with periodic inspections. School of Science, Aalto University, Espoo, Finland, 2021.

J. Leppinen, A. Punkka, and T. Ekholm. A dynamic optimization model for maintenance scheduling of a multi-component system. *Preprint submitted to European Journal of Operational Research*, 2021.

M.L. Littman, T.L. Dean, and L.P Kaelbling. On the complexity of solving markov decision problems. *arXiv preprint arXiv:1302.4971*, 2013.

M.L. Puterman. *Markov Decision Processes, Discrete Stochastic Dynamic Programming.* John Wiley and Sons, Hoboken, New Jersey, 1994.

Y. Shang. A distributed memory parallel gauss–seidel algorithm for linear algebraic systems. *Computers & Mathematics with Applications*, 57(8):1369–1376, 2009.

O. Shlakhter. *Acceleration of Iterative Methods for Markov Decision Processes.* PhD thesis, Graduate Department of Mechanical and Industrial Engineering, University of Toronto, Toronto, Canada, 2010.

D. Wingate, K.D. Seppi, and S. Mahadevan. Prioritization methods for accelerating mdp solvers. *Journal of Machine Learning Research*, 6(5):851–881, 2005.