
ulm university universität
**u**ulm

**Fakultät für Mathematik und Wirtschafts-wissenschaften**
Institut für numerische Mathematik

# C++: Solving Markov Decision Processes

Prüfungsleistung für die Universität Ulm

**Vorgelegt von:**
Adrian Lauber
adrian.lauber@uni-ulm.de
1042606

**Gutachter:**
Prof. Dr. Andreas Borchert

2020

Fassung August 11, 2020

© 2020 Adrian Lauber

Satz: PDF-LaTeX 2$_\varepsilon$

# Contents

# 1 Introduction

The goal of this project is to implement a library to solve *Markov decision processes (MDPs)*. The focus is on extensibility, facilitating modern C++ features like smart pointers as well as following well-established C++ guidelines and best-practices.

The first chapter is a brief introduction into the problem class for which the library functionality is developed.

In the second chapter the focus is on the actual implementation. The class hierarchy is described as aspects of the build and test environment.

# 2 Markov Decision Process

Markov Decision Processes describe a time-discrete and stochastic process that can be used to model different decision-making problems. MDPs are used in all kinds of domains and multiple algorithms have been proposed to solve a particular MDP.

## 2.1 Definition

A MDP is defined by a state-transition-function $P_a(s, s')$ that gives the probability of transferring to a consecutive state $s'$ when in state $s$ and applying action $a$. For a given state $s$ and action $a$ the probability is independent from former states or actions. This property is referred to as the *Markov property*. The goal of optimization is a scalar value referred to as the *cost* or *reward* which is either minimized (in case a problem defined by costs) or maximized (for rewards). The reward/cost is a function of the current state $s$ and consecutive state $s'$ given an action: $R_a(s, s')$

Solving a MDP requires to find a mapping from each state to an action that maximizes reward or minimizes the costs. This mapping is referred to as the *policy* and is denoted by $\pi$.

## 2.2 Algorithms

For finite state and action spaces algorithms based on dynamic programming are a popular choice for solving MDPs. The foundation for applying dynamic programming to this problem class is the definition of a recursive function that assigns a value to a state given that a specific policy $\pi$ is applied.

$$V_\pi = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right] \tag{2.1}$$

Equation 2.1 is referred to as the *Bellmann equation for value functions*. $\gamma$ is the *discount rate* parameter: $0 \leq \gamma \leq 1$

A common algorithm is referred to as *policy iteration*. In an iterative cycle the value function of each state following a policy is first evaluated (*policy evaluation*) and then the policy updated (*policy improvement*). This process is depicted in Figure 2.1.

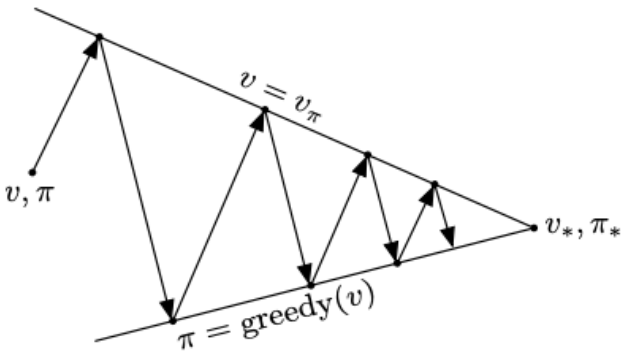


Figure 2.1: Iteration between policy evaluation and policy improvement [2]

# 3 Implementation

## 3.1 Model

The model class is the core element of library because it holds information about the MDP to be solved. It is initialized using the default constructor and a Parser-Module (section 3.2) is parsing the MDP definition from the input file to the model class. Since memory for the model class members has to be allocated dynamically, the parsing model class object offers a public method to allocate the required memory for the object. Once the Parser-Module has determined the size of the state and action space, *SetArrays()* can be called to initialize the array data structures of the state transition matrix and the reward matrix. A method to validate the model data is offered by the model class and is called by the Parser-Module to abort parsing if an inconsistency has been detected.

//MODEL UML

## 3.2 Modules

In order to achieve a high level of extensibility this a base class acts a blue print for classes to read or write the model. Using this common interface enables easy integration which is described in detail in section 3.3. The module class consists of a protected constructor to set its only member: a reference to the model.

The abstract *Parser* class acts as an interface for concrete Parser implementations. There exist several file formats to describe MDPs. For the sake of a running example, a rudimentary implementation to parse *POMDP* file format [1] is provided.

The abstract *Policy* class as a interface and base class for different variations of *policy improvement*. The *policy mapping* member is an Array3D object in order to both support deterministic policies and stochastic policies. A method is provided to select an action given a certain state. This is needed by Evaluation-implementations. The policy class is also used as a return type for the external interface of the library. This way an external application can use the reference to directly integrate the optimal strategy for the environment. The simplest form of policy improvement where always the most promising action is selected has been implemented.

The abstract *Evaluation* class is also both interface and base class for different ways to evaluate a policy. A simple form of policy evaluation is provided in the library: *iterative policy evaluation*. This method applies Equation 2.1 to update the value of each state.

The *Solver* interface and base class provides a general solve method. Solving algorithms combine policy evaluation and policy improvement to find the optimal policy. *policy iteration/monte carlo planning* is implemented as an example.

## 3.3 Module factory

In order to minimize the amount of effort to integrate for example an additional Parser or Solver, a factory is implemented. This enables integration without adding any references in existing code. To ensure scalability for

For each new module two classes are necessary.

- Class with new functionality, inheriting from Module Class
- Constructor class, inheriting from Constructor template

### 3.3.1 Constructor

Each new module

### 3.3.2 Factory

## 3.4 Array3D

To process the state transition matrix and reward structure of a MDP, a multidimensional array implementation is needed. One option is to use *std::vector* holding another std::vector for the two-dimensional case. The std::vector class manages its own resources following RAII (Resource Acquisition Is Initialization)making this a simple and robust solution but performance-wise there is a drawback. The memory for this nested structure will be fragmented which will slow down access. In order to make use of std::vector without causing fragmented memory the *Array3d* template class acts as a wrapper to a one-dimensional std::vector. The () operator accepts three indices and maps those to the 1d std:.vector member. The wrapper performs a boundary check on the 3d indices. If successful, the access to the 1d vector can be performed without the need for a boundary check. This further enhances the performance.

## 3.5 Build

## 3.6 Testing

# 4 Extensibility

# Bibliography

[1] *POMDP file format*. 2020. URL: http://pomdp.org/code/pomdp-file-spec.html.

[2] Richard S. Sutton and Andrew Barto, eds. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. Cambridge, Mass. [u.a.]: MIT Press, 2018, XVIII, 322 Seiten. ISBN: 9780262039246.