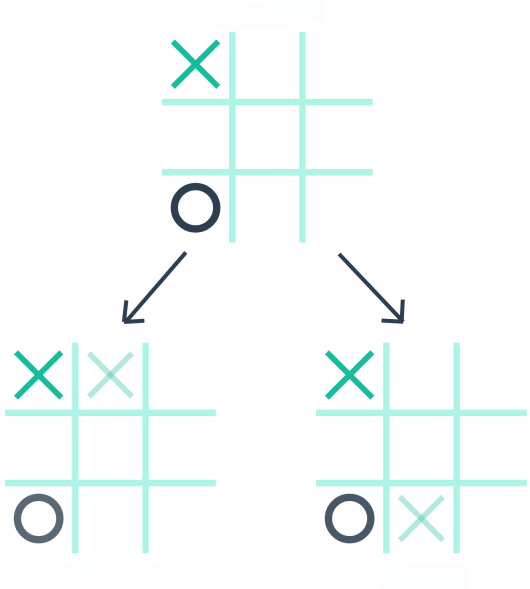
Reinforcement Learning and surrogate reward functions based on graph Laplacians



A MASTER THESIS IN MATHEMATICAL SCIENCE
UTRECHT UNIVERSITY

AUTHOR
Iris Smit

SUPERVISOR
Prof. dr. ir. J.E. Frank

SECOND READER
Prof. dr. R.H. Bisseling

Utrecht University
April 2022

Abstract

Reinforcement learning is an upcoming area in machine learning with many applications. This thesis covers the basics of reinforcement learning: reward functions, value and policy iterations, and their algorithms. A value iteration algorithm for the game tic-tac-toe is given along with the results of a policy learning from itself. When the reward function is not straightforward to define, a surrogate reward function might be helpful. A surrogate reward function is defined by using the Fiedler vector of the Laplacian of the graph defined by the game. Laplacians based on weighted graphs in four different ways are defined and used to make different surrogate reward functions for a walking game. Finally, the surrogate reward functions are used in a value iterations algorithm and compared to the exact value function of the walking game.

Preface

In March of 2016, AlphaGo was the first computer ever to beat the world's best human Go-player: Lee Sedol. AlphaGo won with four against one. Most scientists thought it would have taken at least another ten years to accomplish this. Go is considered the most complex board game for computers to solve. For example, it is exponentially more advanced than chess or checkers.

However, AlphaGo was still being 'raised' by humans, which means that AlphaGo consists of neural networks trained against millions of Go-games of advanced human players. So AlphaGo analyzed the move sequences leading to a win and learned in this way what good and bad moves are, which is called supervised learning.

In October 2017, the AlphaGo Zero was launched, and it beat the AlphaGo by 100 against 0. Surprisingly AlphaGo Zero requires less computing power and has a more straightforward structure than AlphaGo. The AlphaGo Zero consists of only one neural network that started to play Go games against itself. The only human input was the game's rules, including the goal: conquering more area on the board than the opponent. AlphaGo Zero started by making random moves and investigating the sequences of actions leading to a win (for itself or the opponent). Then, after each round, it reprogrammed itself to prefer (sequences of) moves leading to winning. This way of learning is called reinforcement learning.

In the long run, reinforcement learning produces better results than training with good human moves, as the AlphaGo did. D. Silver, J. Schrittwieser, and K. Simonyan showed this in 2017 by resetting the AlphaGo Zero and letting it learn supervised. As seen in figure (0.1), reinforcement learning starts at a disadvantage but quickly surpasses the human level after just 30 hours of training, something supervised learning could never accomplish.

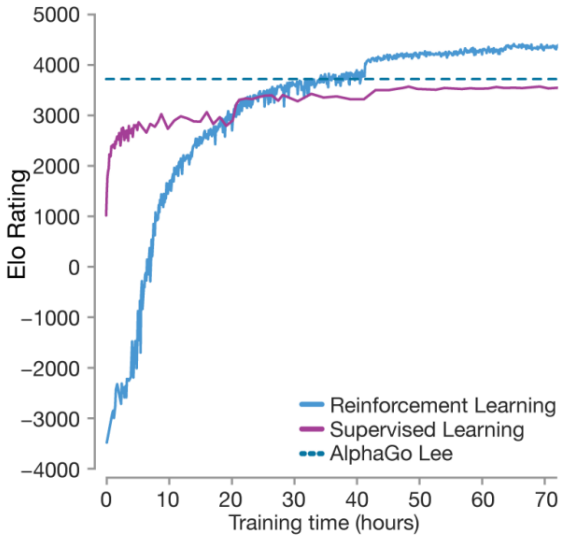


Figure 0.1: Empirical evaluation of AlphaGo Zero (D. Silver et al. 2017)

The first chapter of this thesis is dedicated to explaining different types of machine learning and detailed comparisons to reinforcement learning. Throughout this project, the player or learner of the game is the agent, while everything outside the agent is called the environment. Besides these two core parts, reinforcement learning requires three more fundamental components: the policy, reward, and value function. To start, the main ideas of these three components are introduced, and further in the chapter, a mathematical explanation is given. Throughout this chapter, the game tic-tac-toe illustrates how the different mechanics of reinforcement learning work.

Firstly, the general rules, state, and action space are explained, followed by an in-depth analysis of the reward function. This reward function can be maximized by calculating expected returns while also considering a discount rate. Secondly, the contents of the states in the environment are discussed to establish what information should or should not be contained. Thirdly, both the values of a state and the game's policy are discussed. Several examples demonstrate how the value and policy can be interpreted. Once the general idea has been clarified, the mathematical definitions of both are given, allowing a deduction that leads to the optimal value function and policy (i.e., Bellman optimality equation)—followed by the algorithms for value iteration and policy iteration. Lastly, the chapter extensively explains the game tic-tac-toe and shows how reinforcement learning can be applied to this game. The specific algorithms are given and tested. The player who learns from an imperfect player wins 68% of the time, loses 12%, and ties 30%, whereas the player who learns from itself plays 100% a tie.

The second chapter explores a way to make a reward function for a game where a walker walks on the grid. There are walls on the grid, but the walker does not know beforehand where these are. The reward function is defined by calculating the Laplacian of the graph representation of the grid. Firstly, definitions of directed, undirected and weighted graphs are given, followed by four different definitions for the graph Laplacian and the algorithm for the transition probability matrix of the graph. Secondly, the value iteration algorithms for the sparse reward function (a reward of 1 when winning and 0 otherwise) and the value iterations algorithm when the reward is based on the Laplacian are explained and given. The value function for the sparse reward is calculated exactly and used as a basis to compare the value function for the four different Laplacians. Finally, the results for the different Laplacians are given. First for the probability that going in each direction is equal and second for seven different direction probabilities. All Laplacians converge faster and have a shorter path from start to finish than the sparse reward function. When the directions have different probabilities, one of the four Laplacians, L_{wu}, catches the eye. L_{wu} terminates in the shortest time and always finds the minimum path from start to finish.

Contents

1 Reinforcement Learning

1.1 Introduction Reinforcement learning

Reinforcement learning is a type of machine learning that enables an agent to learn in an interactive environment through trial and error, using feedback from its actions and experiences (Bhatt, 2018). One of the most famous examples of reinforcement learning is the AI program AlphaGo, which used reinforcement learning to beat the world's best players in the board game Go. In October 2015, AlphaGo became the first algorithm to beat a human professional Go player on a full-size 19 by 19 board without handicap. AlphaGo was trained exclusively by playing against itself without any human knowledge, except for the game's rules, Nair (2017) reported.

In 1979 Barto and Sutton started exploring the research area known as reinforcement learning. Reinforcement learning is extensively used in machine learning, neural networks, and artificial intelligence research areas. Since 1979, the mathematical foundation for reinforcement learning has been firmly constructed.

This chapter explains and applies the different aspects of reinforcement learning. First, a brief overview of machine learning and the concept of reinforcement learning is given. Furthermore, the general idea behind reinforcement learning algorithms is explained. Section (1.2) explains the mathematical concepts of reinforcement learning and the formal algorithms following from this, and these algorithms are applied to the board game tic-tac-toe in section (1.3).

1.1.1 Machine learning and its types

Machine learning is a research area of artificial intelligence that develops algorithms and techniques for computers to learn. Unsupervised, supervised, and reinforcement learning are three types of learning in which we can subdivide machine learning. Unsupervised learning is used when data is not labeled and patterns or hidden structures must be found. For example (based on Douglas, 2021), a data set could contain animal physical and social features. Unsupervised learning can group or cluster the animals based on similar features like giving birth, laying eggs, hair, or feathers.

Models with *supervised learning* algorithms take data as input and have the goal of labeling the data correctly. Continuing with the animals' dataset example: the physical and social features of the animals can be used as input data; the algorithm then has the goal of labeling which species belong to which features. First, the model will need some correct animal features - species pairs. Then the model tries to guess the species' new features and needs a supervisor to tell if this is correct. If the algorithm guesses the species wrong, the supervisor can tweak the model to guess more accurately next time. Finally, when the model is trained with enough data, it can match the unlabeled features with the most probable species (Douglas, 2021).

Reinforcement learning (RL) is different from (un)supervised learning because it does not use static data to cluster or label the data in any way. Instead, reinforcement learning uses dynamic data obtained by the environment to maximize a numerical reward. In models where RL is used, there is an agent (i.e., player of a game) who can take an action (i.e., make a move) in each possible state (i.e., board configuration of the game) to change

the state (into a new board configuration). After each action, the agents receive a reward, used to learn which action to take and not to take in the future. The agent tries to maximize the total reward in the long run by choosing the best sequence of actions.

In a way, humans learn in the same way as reinforcement learning does. One can think of a human as an agent, the earth as the environment where the agent can interact with and get rewarded for making actions. If a student finishes her master's degree (action), she finds a job (state), and the job pays well (reward), she got positively rewarded by performing a good action. On the other hand, if the student stays up late the night before an exam (action), she is tired the following day (state), and she fails the course (reward). Hence she got negatively rewarded for a wrong action.

1.1.2 Reinforcement learning algorithms

Apart from the agent and the environment, a reinforcement learning system has three fundamental ingredients: a policy, a reward, and a value function. The *policy* is a mapping that can be seen as the strategy according to which a player plays. The player watches the opponent make a move (the input), and the strategy tells him which move is best to make next (output). The highest goal is to find an optimal strategy to beat all players. Reinforcement learning algorithms come in when the current policy is not yet optimal. A strategy cannot be optimal when a player starts playing and is not an advanced player yet. Alternatively, the environment might be changing because the opponent changes strategy. Either way, the policy needs to be changed to meet the agent's goal: find the sequence of actions that maximize the reward in the long run (Douglas, 2021). The student of the previous paragraph who failed her course will (hopefully) adjust her policy to not stay up late for the next exam.

In a reinforcement learning problem, the reward signal defines the goal. The reward signal tells the agent if the action taken was a good or a wrong choice. The agent aims to gather as much reward as possible in a certain period. However, the agent should only be rewarded for the main goal, not subgoals. For example, a chess player should only be rewarded for winning and not gaining pieces from the other player. Otherwise, the player might find a strategy to win all the pieces of the other player (subgoal) instead of winning the game (actual goal). The *reward* is a numerical value the agents receive for being in a specific state (Douglas, 2021). Perhaps an action to a state that gives a low reward now might result in a bigger reward in the future. Hence, next to the reward, there is a *value* of a state which is the total reward that the agent can expect from that state (taking into account the long term). If the agent, instead of looking at the reward received at a state, looks at the value, this will help choose a more beneficial action in the future. However, the high reward predicted in a future state might not be there anymore once the state is reached. In RL, future reward prediction is taken into account by a *discount factor* which decreases rewards further in the future (Watkins, 1989).

When using a reinforcement learning algorithm, besides the discount factor, another decision needs to be made: in what proportion exploration and exploitation of actions are made. When a reinforcement learning agent uses *exploitation*, he only chooses actions he has already made before. The agent knows the rewards of past actions and can use this knowledge to maximize the total reward. However, exploring new areas of the environment (*exploration*) is necessary as well (Douglas, 2021). For example, if a woman wants to decide which restaurant she wants to eat at, she chooses a restaurant she has already been to, exploiting her knowledge that the food was good. Alternatively, she explores a new restaurant, where she has not been before, increasing her knowledge. She might get disappointed with the food; however, this approach could also potentially find her a new favorite restaurant. Finding a balance is key, and luckily reinforcement learning

algorithms provide a way of settling that balance.

Three things must be maintained if one has a model suited for a reinforcement algorithm to find an optimal strategy. Firstly the policy needs to be defined with the correct number of parameters and structure. Secondly, the reward function needs to be defined such that the model knows when a successful action has been made. Thirdly, an efficient algorithm needs to be chosen which applies the reward function correctly to each state and knows how the parameters need to be tweaked. Also, the discount factor, exploration, and exploitation parameters must be set (Douglas, 2021).

1.2 Reinforcement Algorithm

The reinforcement learning algorithm is mathematically defined in this section through the famous game tic-tac-toe. First, the general rules, reward, and action space are explained. Secondly, the sequence of rewards is presented, the returns are defined, and the Markov property is made clear. After this, the fundamental parts of reinforcement learning algorithms are described: the value function and policy, followed by the corresponding value and policy iteration algorithms. The final paragraph applies the value function algorithm to tic-tac-toe, and results are shown.

1.2.1 Tic-tac-toe: general rules, reward, state, and action space

Tic-tac-toe is played on a three-by-three board where two players take turns. One player plays Xs, and the other player plays Os. The game is finished when a player has three marks in a row (vertically, horizontally, or diagonally), the player who accomplishes this is the winner. The game can also end when the board is full, but there are no three marks in a row - which means there is a tie. The learning agent in this example is player X, and everything the player interacts with is called the environment. The interactions are the moves (actions) the player chooses, the numerical rewards belonging to these moves, and the new situations the player faces when the opponent makes a move. Each move, either by X or by O, is seen as a discrete-time step $t = 0, 1, 2, \ldots, 9$.

The state space is defined as $\mathcal{S}$. Per time step, the agent receives a representation of the environment's state $S_t \in \mathcal{S}$. The environment's state s_t is abbreviated as s in cases where the context makes this clear. For example, the vector $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ represents the playing board in the following way:

$$\begin{array}{ccc} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{array}$$

Each position of the board can be either empty (0), has an X (1), or an O (2). The board in figure (1.1) is represented as $S_t = [1, 2, 0, 0, 1, 0, 2, 1, 2]$. Since each position has three possible values, the cardinality of $\mathcal{S}$ is 3^9. However, not each of the 3^9 states will be feasible; this is described at the end of chapter one (see box (4.3)).

Figure 1.1: Tic-tac-toe

The available actions in state S_t at time step t is defined as the set $\mathcal{A}(S_t)$. An action $A_t \in \mathcal{A}(S_t)$ is possible in each empty square of the board (indicated by a zero). The action is abbreviated to a in cases which the context makes this clear. For the state represented in figure (1.1) an X can be placed in the third, fourth or sixth square, therefore $\mathcal{A}(S_t) = \{3, 4, 6\}$. An action is represented as a nine digit vector of all zeros with a one for player X (and a two for player O) in the position the player wants to makes a move. Take for example $A_t = 3$, this is represented as $[0, 0, 1, 0, 0, 0, 0, 0, 0]$. After making move $A_t = 3$ the player finds him/herself in a new state $S_{t+1} = [1, 2, 1, 0, 1, 0, 2, 1, 2]$. Like Barto and Sutton (2014) another consequence of the players action, one time step later, is receiving a numerical reward.

The reward is dependent on whether the learning player is X or O. Assume the learning player is X without loss of generality; he wins when there are three Xs in a row and receives a reward of 1. When there are three Os in a row, he loses and receives a negative reward of -1. If the game has not ended and neither party has three in a row, the learning player receives a zero reward. When all squares are filled, and neither player has won, there is a tie and the reward received is 0. The reward is, similar to Barto and Sutton (2014), defined as $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ with for tic-tac-toe the following specific values:

$$R_{t+1} := \begin{cases} 1 & \text{if player X wins in time step t+1} \\ -1 & \text{if player O wins in time step t+1} \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

1.2.2 Returns

The goal of the agent is to maximize the reward after some time. In equation (1.1), the direct reward of being in a specific state has been defined. Consider the sequence of rewards received after time t: $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$. To maximize the reward in the long run, the *expected return* has to be maximized. Barto and Sutton (2014) define the return $\mathcal{G}_t$ as an explicit function of the reward sequence, where T is considered as the final time step. In the straightforward case, Barto and Sutton (2014) define the return as follows:

$$\mathcal{G}_t := R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T \tag{1.2}$$

Definition 1.2 only makes sense when there is a clear final time step. Barto and Sutton (2014) divide tasks into *episodic tasks* which have a clear begin and endpoint and *continuing tasks* which go on without limit (i.e., $T = \infty$). Continuing tasks require a different return function that is out of this thesis's scope. In reinforcement learning, tasks are subdivided into finite-horizon, indefinite-horizon, and infinite-horizon. Otterlo and Wiering (2012) explain that finite-horizon tasks are episodes with a finite number of steps and indefinite horizon tasks are episodic tasks that end but can have arbitrary length. Infinite horizon tasks do not end at all like continuous tasks. In this thesis, only episodic tasks are further explored, and a deeper division is not needed to explain the theory of the reinforcement learning algorithms. Tic-tac-toe is an episodic task where the beginning is an empty board, and the endpoint is when a player has won, lost, or tied.

Discount rate

The prediction of a high reward in the future is less reliable, and therefore that reward might not be there by the time it is reached. Watkins (1989) was the first to take this into account by a *discount rate* defined as γ, which decreases rewards further in the future. The discount rate is a parameter with $0 \leq \gamma \leq 1$. Barto and Sutton (2014) use the

discount rate to define the *discounted return*:

$$\mathcal{G}_t := R_{t+1} + \gamma \mathcal{G}_{t+1} = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}, \text{where } \mathcal{G}_T = R_T \tag{1.3}$$

When the discount rate equals zero, the agent is only bothered by direct rewards and discards the long-term return. He learns which A_t to choose resulting in maximum R_{t+1}. However, the total return will probably be reduced in comparison with $\gamma > 0$. If the discount rate approaches one, the agent takes future reward more strongly into account, and $\mathcal{G}_t$ will have a finite value. If $\gamma = 1$, the agents consider all future rewards equal to the immediate rewards.

Discount rates in reinforcement learning algorithms are only used for episodic tasks. Even in episodic tasks, Mahedavan (1996, p.160) shows that using a discount rate can lead to sub-optimal behavior if a mediocre payoff solution looks short-term more attractive than long-term high rewards. There are deep programming algorithms for finding optimal average reward policies, but these algorithms need information about the complete transition probability matrix (White, 1963, Howard, 1960). Transition probability matrices can have a significant computational factor or are impossible to define (i.e., when the opponent's strategy is unknown). Hence, this thesis uses the discount return with the discount rate as described in equation (1.3).

1.2.3 Markov property

The policy tells the agent which action is best to make, based on the agent's state. This section explains which information the states of the environment should and should not contain. The state is a description of the environment at the current time. The state should entail all the agent's information learned in the past but can compactly do this. For example, take a chess game where the agent is a chess player, and the state is the current board configuration. A board configuration tells where all the pieces are currently positioned, and this also implies which pieces have already been captured. A board configuration does not describe in what exact order pieces have been moved or captured, but this is not important for the game's future.

Barto and Sutton (2014) explain that a game from which the states entail all information such that looking to previous states is unnecessary for predicting the future is said to have *the Markov property*. The checkers game described above has the Markov property. Durret (2019) formally defines the Markov property as:

Definition 1.1. If the state space X is countable, **the Markov property** holds if for any states $i_0, \dots i_{n-1}, i$ and j:

$$\mathbb{P}(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, \dots, X_0 = i_0) = \mathbb{P}(X_{n+1} = j | X_n = i).$$

Since the state space, X is countable, working with sums and probabilities is possible. However, if the state space is infinite, integrals and probabilities are needed (which is out of the scope of this thesis).

Definition 1.2. A **Markov decision process** (MDP) is a reinforcement learning task that satisfies the Markov property as defined in 1.1. It is called a **finite Markov decision process** if the state and action space are finite.

The Markov property holds if the probability of transitioning from one state to another (i.e., $p(i,j)$), is equal to $\mathbb{P}(X_{n+1} = j | X_n = i)$ (Durret, 2019). A game is a Markov decision process when its state and action sets are defined by the one-step dynamics of the environment:

$$p(s', x|s, a) = \mathbb{P}(R_{t+1} = x, S_{t+1} = s'|S_t, A_t), , \tag{1.4}$$

where p is the probability from transitioning from a given state s and action a to each possible pair of next state s' and reward x, and $\mathbb{P}$ is the probability distribution of the environment's dynamics.

Tic-tac-toe is a finite Markov decision process. Finite because it has a finite state space $|\mathcal{S}| = 3^9$ and finite action space $|\mathcal{A}| = 9$. It is a MDP because its states satisfy the Markov property, as can be seen in the following example:

Example 1.1. *Recall the tic-tac-toe game in figure (1.1) with the following state and action:*

$$s = [1, 2, 0, 0, 1, 0, 2, 1, 2]$$
$$a = [0, 0, 1, 0, 0, 0, 0, 0, 0].$$

Assume the opponent always plays a random move, in this case on the fourth square, so the new state is $s' = [1, 2, 1, 2, 1, 0, 2, 1, 2]$*, with a reward of zero. Hence the following holds:*

$$p(s', x|s, a) = \begin{cases} 1 & \textit{if s'=[1,2,1,2,1,0,2,1,2] and x=0} \\ 0 & \textit{otherwise} \end{cases} \tag{1.5}$$

which is equal to $\mathbb{P}(R_{t+1} = x, S_{t+1} = s'|S_t, A_t)$*, this holds for all states and hence tic-tac-toe is a MDP.*

If a game is a MDP and the dynamics are specified as in equation (1.4) it is possible to predict the likelihood of the next state and next reward, given the current state and action. Iterating equation (1.4) shows, from the current state-action pair, the expected reward can be predicted (Barto and Sutton, 2014)):

$$r(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{x \in \mathcal{R}} x \sum_{s' \in \mathcal{S}} p(s', x|s, a) \tag{1.6}$$

Likewise the *state-transition probabilities* can be calculated,

$$p(s'|s, a) = \mathbb{P}\{S_{t+1} = s'|S_t = s, A_t = a\} = \sum_{x \in \mathcal{R}} p(s', x|s, a) \tag{1.7}$$

and the expected rewards for state-action-next state triples,

$$r(s, a, s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] = \frac{\sum_{x \in \mathcal{R}} x p(s', x|s, a)}{p(s'|s, a)}. \tag{1.8}$$

Hence the full dynamics of a finite Markov decision process can be described.

When a game is not a MDP there is a state and action pair which don't satisfy the Markov property and may depend on all the previous states and actions. Barto and Sutton (2014) explain that the general response of the environment at time $t + 1$ to an action taken at time t, can only be described by the complete probability distribution:

$$\mathbb{P}(R_{t+1} = x, S_{t+1} = s'|S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t) \tag{1.9}$$

for all possible values of events $S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$ and all x, s'. In particular, when a game is not Markov: equation (1.9) is not equal to (1.4) for all s', x and all events $S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$.

In reinforcement learning, the value function is assumed to depend only on the current state; hence the Markov property must hold for the environment. Barto and Sutton (2014) state that even when a game is not a MDP, the game can be thought of as an approximation to a MDP. When the Markov property does not strictly apply, the theory represented after this section does not strictly apply in the same way. Nonetheless, non-MDP games are out of the scope of this thesis.

In the tic-tac-toe game, the goal is to train the learning player to play optimally. The player is only provided the rules of the game and the rewards. Formally this means the algorithm can only make a move (place a 1 or 2) in an empty square (denoted by a zero in the vector) when it is its turn. The final paragraph explains how the algorithm knows if it is its turn by a turn function, and the code is given. The rewards are given through the reward function as defined in (1.1). The algorithm needs to know when a player wins to use the reward function. The algorithm is given a function that determines if a player wins and is also explained in the final paragraph. So the player knows how to play the game and *what* to achieve (winning) but not *how* this can be achieved. The following sections explain how the probability of winning can be found and how to make a policy. In section (1.3.6), the learning player plays against an imperfect player to set up the first values and a policy. In section (1.3.7), the learning player is improved by playing against itself and thereby learning from itself.

1.2.4 Value function and policy

The agent's goal is to find an optimal policy, in other words, determine the best action to be taken in each state. Therefore, a specific measure is introduced to determine if one action is better than another. The measure used to establish how good it is to take a particular action in a state is called the *value*. This section shows how each action-state pair's value is set and how these values are used to make a policy.

Value

In reinforcement learning, the optimality of an action in a given state is the probability of winning from the subsequent state (Barto and Sutton, 2014, Otterlo and Wiering, 2009 and Torres, 2020). Shannon (1950) already developed the idea of giving a state a numerical value representing the long-term benefits. Shannon (1950) suggested using a value function for programming a computer to play chess. A human comparison of the reward is a high reward if someone feels pleasure and a low reward if someone feels pain. In the same line, the value of a state can be compared with a long-sighted view how satisfied or unsatisfied someone is in his/her environment in a specific state. In this thesis, the probability of winning from a state is defined as the value of the state: $V(s)$. The following example, inspired by Jing (2019), gives an insight into values at a tic-tac-toe game.

Example 1.2. *In figure (1.2), two states of a tic-tac-toe game are shown. In state A, both players have not won yet; hence, the reward is zero. However, state A is only one move away from state B, where player X wins. Hence being in state A should have a good value, even though there is no victory yet.*

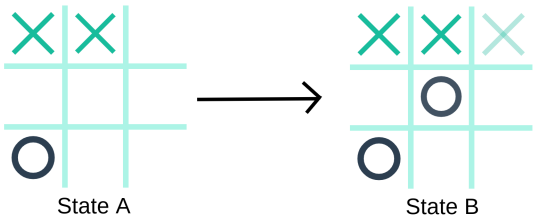


Figure 1.2: Tic-tac-toe states A and B

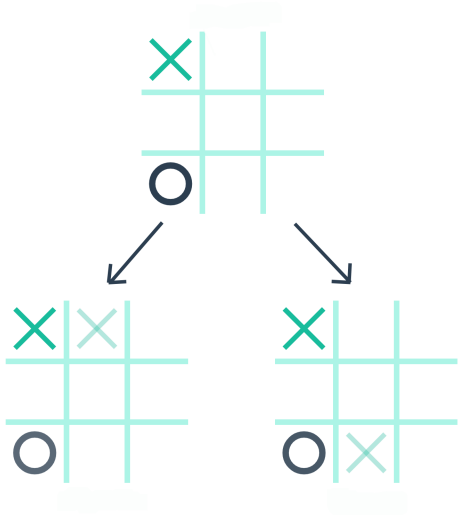


Figure 1.3: Tic-tac-toe states C, A and D

Assume a new learning round is started and player X is considering how to make her next move, as shown in figure (1.3). The left option brings her in state A, and the right choice brings her in state D. Since state A brings her closer to winning, she makes the left move. Because state C can bring player X in state A, state C should get a fraction of state A's (good) value.

In reinforcement learning algorithms, estimates of the values are made when playing a game round. After each round, more precise estimates of the probability of winning from each state can be made. In example (1.2) is seen that in the new learning round, the player has gathered more information (state A is good), and hence a new estimate of the value of state C can be made. Before giving the formal definition of the value function, the policy is explained.

Policy

The player's policy is the strategy according to which a player plays. The policy can be a lookup table or a simple function in some games. In other games, the policy might be stochastic. In this thesis, the policy π is defined in the following way:

Definition 1.3. The *policy* π is the probability $\pi_t(a|s)$ of choosing a specific action $A_t = a$ given the player is in state $S_t = s$ at time step t. There, A_t is the action space and S_t is the state space at a specific time step where $A_t \subseteq \mathcal{A}$ and $S_t \subseteq \mathcal{S}$.

The policy and the value function are closely related. More precise estimates of the values can be made when more learning rounds have been made. Hence, the policy can

be changed into an optimal policy in which the player receives a maximum reward. In tic-tac-toe, the policy is a rule that tells the player what move to make in every possible state. For example, a policy can be: always make your move in the corner squares or constantly try to block your opponent. In addition, if an agent is in state zero of figure (1.3), policies can be:

π_a The agent always goes left

π_b The agent always goes right

π_c The probability of going left and right is both 0.5

π_d The probability of going left is 0.3 and going right is 0.7

To further clarify the relationship between the policy and value, take a look at the following example based on Torres (2020):

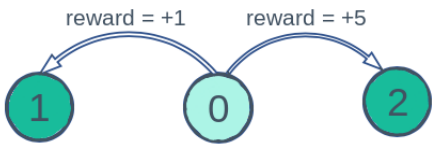


Figure 1.4: Transition diagram example (1.3)

Example 1.3. *The state-space consists of three states, and the agent's beginning state is state zero. States one and two are the terminal states. Arriving in state one gives a reward of +1, and state two gives a reward of +5. The transition diagram of the states is given in figure (1.4). Consider the policies π_a, π_b, π_c and π_d as given above. The value of state zero, $V(0)$, for the policies is:*

a Every time the agent goes left, he receives a reward of +1 and the episode ends, hence $V_{\pi_a}(0) = 1$

b Every time the agent goes right, he receives a reward of +5 and the episode ends, hence $V_{\pi_b}(0) = 5$

c The probability of going right and left is both 0,5, hence $V_{\pi_c}(0) = 0.5 \cdot 1 + 0.5 \cdot 5 = 3.5$

d The probability of going left is 0.3 and going right is 0.7, hence $V_{\pi_d}(0) = 0.3 \cdot 1 + 0.7 \cdot 5 = 3.8$

Since the agent aims to obtain the highest reward possible, the optimal policy is π_b: always going right (Torres, 2020).

Example (1.3) might give the impression that always taking the action with the highest reward is the best option. However, consider the following example to see that this is not always the case.

Example 1.4. *Let's extend the previous example by adding state three with a reward of -8. The state transitions with rewards are represented in figure (1.5) State two is no longer a terminal state but a transition state to state three, which is a terminal state. Calculating the value of state zero for the different policies of example (1.3) gives:*

a Always left is the same: $V_{\pi_a}(0) = 1$

b Always right involves now two steps until termination: $V_{\pi_b}(0) = 5 + -8 = -3$

c $V_{\pi_c}(0) = 0.5 \cdot 1 + 0.5 \cdot (5 + -8) = -1$

d $V_{\pi_d}(0) = 0.3 \cdot 1 + 0.7 \cdot (5 + -8) = -1.8$

The best policy is no longer always going right; in fact, policy π_a*: always going left is now the optimal policy (Torres, 2020).*

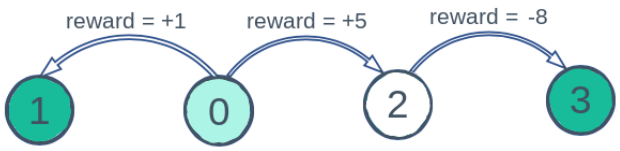


Figure 1.5: Transition diagram example (1.4)

Value function and policy

The value of a state s under a policy π, is the expected return when starting in state s and choosing all the actions the policy π suggests to make. For Markov Decision Problems Barto and Sutton (2014) define the value of a state according to the return $\mathcal{G}$ and policy π in the following way:

$$v_\pi(s) := \mathbb{E}_\pi[\mathcal{G}_t | S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right]. \tag{1.10}$$

Here, $\mathbb{E}_\pi[\cdot]$ is the expected value of a random variable, given that the agent follows policy π and the time current time step is t. The state in which a game is ended by winning, losing, or playing a tie is called a terminal state. The value of a terminal state is always equal to zero.

When a player is playing according to a specific policy π where the values v_π are estimated by saving all the returns gathered from a state and, finally, averaging all the returns. The average of all the returns will converge to the actual value of the state $v_\pi(s)$ when the number of times the state has come across goes to infinity. This way of averaging over many random samples of returns is called a *Monte Carlo method*. The first time Monte Carlo methods were used to estimate action values in reinforcement learning was by Michie and Chambers (1968). It may be impractical to keep separate averages for each state if there are many states. In this case, the agent can maintain a parameterized function of v_π and adjust the parameters as more returns are observed, which is in the reinforcement learning setting represented by Bertsekas and Tsitsiklis (1996) but lies out of the scope of this thesis. In tic-tac-toe, we have $|\mathcal{S}| = 3^9$, which is suitable for the Monte Carlo method.

For any policy π, the value of s and its possible successor states satisfy the following specific recursive relationships (Barto and Sutton, 2014):

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[\mathcal{G}_t|S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s\right] \\ &= \sum_a \pi(a|s)\sum_{s'}\sum_x p(s', x|s, a)\left[x + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s'\right]\right] \\ &= \sum_a \pi(a|s)\sum_{s',x} p(s', x|s, a)\big[x + \gamma v_\pi(s')\big] \end{aligned} \tag{1.11}$$

Notice that expression (1.11) is an expected value, since it is a sum over all values a, s' and x, the probabilities $\pi(a|s)p(s', x|s, a)$ are computed, $x + \gamma v_\pi(s')$ is weighted by that probability and summed over all possibilities. This last equation is called the *Bellman equation* and is one of the central elements in many reinforcement learning algorithms. Richard Bellman (1957) was the first to introduce equation (1.11), which was named the "basic functional equation" back then. The continuous case of the Bellman equation is the Hamilton-Jacobi-Bellman equation and descends from classical physics (Shultz, 1967). Equation (1.11) is for the discrete case and simplifies the value function into less difficult, recursive subproblems, which makes finding the optimal value function easier.

Optimal value functions and policies

The ultimate goal of solving a reinforcement learning problem is finding the *optimal policy* denoted by π_*. In the case of finite Markov decision problems, an optimal policy can precisely be determined. Barto and Sutton (2014) state that a policy π is better than or equal to another policy π' when the expected return is greater or equal for all states. In other words:

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S}. \tag{1.12}$$

Continuing, Barto and Sutton (2014) state that an optimal policy is guaranteed to exist but might not be unique. If there is more than one optimal policy (all denoted by π_*), the policies share the same optimal state-value function $v_*(s)$ defined by:

$$v_*(s) := \max_\pi v_\pi(s) \tag{1.13}$$

for all $s \in \mathcal{S}$. Since the optimal value function in (1.13) is the value function for the optimal policy π^*, it also satisfies the Bellman equation (1.11) for all state values. As explained before, a state's (optimal) value under an (optimal) policy is the expected return for the best action from that state. Hence, equation (1.13) can also be written as the maximum over

the expected return, leading to the following set of equations (Barto and Sutton, 2014):

$$\begin{aligned}
v_*(s) &= \max_\pi v_\pi(s) \\
&= \max_a \mathbb{E}_{\pi^*}[\mathcal{G}_t | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi^*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}_{\pi^*}\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} | S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s',x} p(s', x|s, a)\big[x + \gamma v_*(s')\big] \qquad (1.14)
\end{aligned}$$

The last equation is called the *Bellman optimality equation* and considers the actions resulting in the maximum reward, where the Bellman equation considers all possible actions. If there are K states, then the Bellman optimality equation is a system of K equations in K unknowns. Hence v_* can be solved if K is not too big, $p(s', x|s, a)$ is known and the Markov property holds. From the optimal value function, the optimal policy can be determined. Any policy that appoints a non-zero probability only to the actions where a maximum in equation (1.14) is obtained is an optimal policy.

Explicitly solving equation (1.14) is based on at least the following three assumptions: a small state-space, knowing $p(s', x|s, a)$ and the Markov property holds. Seldom all three assumptions are met in practice. For example, the pit and pebble game Awari, whereby two players have six pits called "houses" and 48 seeds in the game. Each turn, a player can remove the seeds from his/her house and leave one seed in each house counterclockwise of the house the seeds are taken from. If the player ends in the opponent's house and two or three seeds are in this house, the player can take these seeds and the seeds of consecutive houses with two or three seeds. The player who captures more than 24 seeds wins the game. In figures (1.6) and (1.7) a move of a Awari game is shown. The downplayer starts with moving his six seeds from house V (marked with a green circle in figure (1.6)) to houses VI, a, b, c, d, and e. Since he ends up with two or three seeds in houses c, d, and e (marked with a green circle in figure (1.7)), he can take the seeds in these houses. For Awari, $p(s', x|s, a)$ is known, and the Markov property holds, but the number of states is 10^{12} and hence the computation time is too large (Allis, 1994). In reinforcement learning, one commonly settles for approximate solutions of the Bellman optimality equation. There are many different decision-making methods to approximate equation (1.11); for tic-tac-toe, this can be done using the dynamic programming method described in the next paragraphs.

1.2.5 Policy Iteration Algorithm

The following two sections explain the policy and value iteration algorithms to find approximate solutions for the optimal value function and policy. Algorithms that compute optimal policies (given a perfect environment model) are a part of dynamic programming. Bellman (1957) invented the term dynamic programming, which simplifies a complicated problem by breaking it down into simpler sub-problems. Classical dynamic programming algorithms are not significantly used in reinforcement learning because they assume a perfect environment model. In a perfect model, the one-step transition probabilities and expected rewards for all states and all actions possible are known, which imaginable is

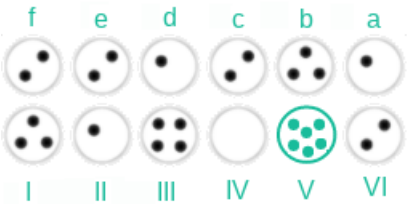


Figure 1.6: Awari board before move

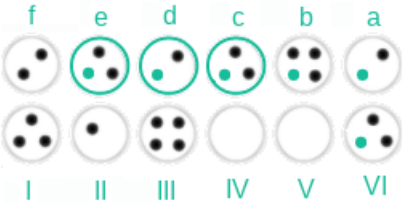


Figure 1.7: Awari board after player one put his seeds from V in VI, a, b, c, d and e.

not always the case. Even if one has a perfect model, the agent can often not perform enough computation per time step to use all the information. Also, memory is an essential constraint since there are, in practice, often far more states than there could be entries in a table; approximations must be made (Sutton and Barto, 2014).

Minsky (1961) was the first to connect dynamic programming and reinforcement learning on a checkers game by solving Markov decision problems. Watkins (1989) explicitly connected reinforcement learning to dynamic programming in a class called incremental dynamic programming. The algorithms considered in the following paragraphs belong to incremental dynamic programming. The algorithms do not assume a perfect model and have less computation time than classical dynamic programming algorithms. At first, the computation of the value function v_π is considered for an arbitrary policy π. Second, improving and evaluating a policy is explained. Finally, the value iteration algorithm, which combines policy iteration and evaluation, is explained.

Policy Evaluation

Policy evaluation is the iterative computation of the value function v_π for a given policy π. Consider the Bellman equation (1.11), which for any policy calculates the value of a state (and its possible successor states). Recall the discount factor introduced at page 9. If either the discount rate γ is smaller than one or for all states termination is guaranteed under π, the value function of π exists and is unique. Since solving the value function exactly may take a lot of computation time, approximate value functions are considered. Write the sequence of approximate value functions as $v_0, v_1, v_2, \ldots$. The approximate value functions map states to $\mathbb{R}$. For the initial v_0, values are chosen arbitrarily, excluding the terminal state, which must be given value 0. For tic-tac-toe, these are the states where one player wins, or there is a tie. Each next approximation is obtained by using the update rule of the Bellman equation for all $s \in \mathcal{S}$ (Barto and Sutton, 2014):

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',x} p(s', x|s, a)\big[x + \gamma v_k(s')\big] \end{aligned} \tag{1.15}$$

The iterative value function v_k has a fixed point, namely the policy value function v_π that satisfies equation (1.11). Barto and Sutton (2014) show that the sequence $\{v_k\}$ defined by (1.15) converges to v_π if the number of iterations k goes to infinity. The convergence

occurs if either $\gamma < 1$ or if the game terminates for all states under policy π (Barto and Sutton, 2014). If one successfully wants to approximate v_{k+1} from v_k: the old value v_k at a state s needs to be replaced with a new value $v_{k+1}(s)$. The replacing new value is obtained from all the old values, and the expected rewards from all the (one-step) successor states of s, all concerning the current policy evaluated. This type of evaluating described is called *full backup* because it is based on all the successor states and not only a subset which is called *sample backups* (Barto, 1995). Hence, combining this into an algorithm needs one array for the old values and one for the new ones. The complete *iterative policy evaluation* algorithm is shown in box 1.1.

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',x} p(s', x|s, a)\big[x + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Box 1.1: Policy evaluation, source: Barto and Sutton (2014)

For a large number of states, iterative policy evaluation suffers Bellman's curse of dimensionality because of the full backup. Hence, in practice one commonly uses

$$\max_{s \in \mathcal{S}} |v_{k+1}(s) - v_k(s)|$$

as a stopping condition, which in box 1.1 is captured in Δ (Barto and Sutton, 2014).

Policy Improvement

The reason for computing the value function for a specific policy is to help improve that policy. After one round of policy evaluation (one sweep through all the states), a value function with respect to the input policy π is calculated. Generally, the optimum is not attained in a finite number of iterations; hence there exists an improvement. Consider policy π and π' and recall line (1.12):

$$\pi \geq \pi' \text{ if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S}.$$

For simplicity, assume that the policy only differs for one state: s. So $\pi'(s) = a \neq \pi(s)$ and for all other states they are equal. If the value in s for policy π' is greater than the value in s for π, then π' is a better policy and hence an improved policy. In state s, it is better to choose action a than the action policy π had in mind for this state. Since the state space can become quite large, Barto and Sutton (2014) defined a function which at each state changes the action into the best action (known) at once:

$$\begin{aligned} \pi'(s) &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \arg\max_a \sum_{s',x} p(s', x|s, a)\big[r + \gamma v_\pi(s')\big], \end{aligned} \tag{1.16}$$

where argmax_a is the value of action a which maximizes the sum. The new policy π' calculated with this formula takes the best action (according to v_π), looking only one

step forward. Hence the policy is *greedy* since it only exploits actions that have been made before. Policy π' improves the original policy π concerning v_π; this is named *policy improvement*. The inequality in line (1.12) must be strict for policy improvement, except when the original policy is optimal. If there is a policy π' which satisfies $\pi' \geq \pi^*$, then both are optimal policies, and they have the same unique value function: $v_{\pi'} = v_{\pi^*} = v_*$ (Barto and Sutton, 2014).

Policy Iteration

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathbb{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',x} p(s', x|s, a)\big[x + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)
3. Policy Improvement
   policy stable $\leftarrow$ true
   For each $s \in \mathcal{S}$:
   $a \leftarrow \pi(s)$
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',x} p(s', x|s, a)[x + \gamma V(s')]$
   If $a \neq \pi(s)$, then *policy stable* $\leftarrow$ false
   If *policy stable*, then stop and return V and π; else go to 2.

Box 1.2: Policy iteration, source: Barto and Sutton (2014)

Naturally after finding a new policy by applying (1.16), this policy needs to be evaluated (box 1.1). If the policy π is still not optimal another round of policy improvement needs to be done by using v_π to get a better policy π'. With the *iterative policy evaluation* algorithm the improved value function $v_{\pi'}$ of the improved policy π' can be calculated. Iterating improvement ($\xrightarrow{\text{I}}$) and evaluation ($\xrightarrow{\text{E}}$) of policies leads to the desired optimal policy and corresponding optimal value function.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_* \tag{1.17}$$

The iterating process of improvement and evaluation converges in a finite number of iterations to an optimal policy and value function (Barto and Sutton, 2014). Not surprisingly, this process is called *policy iteration*, and the complete algorithm is given in box (1.2). Howard devised policy iteration in 1960 after Bellman introduced the Markov decision processes in dynamic programming in 1957. The initialization phase is done once in policy iteration, and policy evaluation and improvement are repeated until an optimal policy is found. An optimal policy is found when no further improvement is possible, i.e., the policy maps each state to the best action, and there are no more actions in the policy improvement round. The previous value function is used in each round and is slightly

modified. However, the value function typically changes slightly from one round to the other. Moreover, policy iteration often needs only a few iterations to find the optimal policy (Barto and Sutton, 2014).

1.2.6 Value Iteration Algorithm

An inconvenience of policy iteration is that each iteration involves a policy evaluation, which may require multiple sweeps through all the states, hence much computation time. When policy evaluation is done iteratively, convergence to an exact value function v_π only occurs in the limit. Fortunately, exact convergence of policy evaluation is not needed (Barto and Sutton, 2014). When each state in policy evaluation is only considered once the algorithm is called *value iteration* as seen in box 1.3. Turning the Bellman optimality equation into an update rule gives value iteration (Barto and Sutton, 2014):

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s',x} p(s', x|s, a)\big[x + \gamma v_k(s')\big] \end{aligned} \tag{1.18}$$

Notice how this equation is obtained from the update rule of the Bellman equation (1.15) used in policy evaluation. The difference is that (1.18) requires the maximum taken over all actions. Thus value iteration combines policy evaluation and policy improvement. In the same way, as policy iteration needs an infinite number of iterations to get an exact optimal policy, value iteration also needs an infinite number of iterations to get an exact optimal value function v_*. Similarly, the algorithm is terminated when the value function only changes a little from one round to another. How much the value function changes is captured in Δ. In each round of value iteration: one round of policy evaluation and one round of policy improvement are combined. Since value iteration combines policy evaluation and improvement, it also converges to an optimal policy for finite Markov decision problems (Barto and Sutton, 2014).

Initialize array V arbitrarily

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',x} p(s', x|s, a)[x + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy π, such that
  $\pi(s) = \arg\max_a \sum_{s',x} p(s', x|s, a)[x + \gamma V(s')]$

Box 1.3: Value iteration, source: Barto and Sutton (2014)

1.3 Tic-tac-toe: value function

Consider the game tic-tac-toe again to get a practical insight into the applications of the value iteration algorithm. The goal is to find an optimal policy (i.e., player X never loses) through value and policy iteration. The value iteration algorithm for tic-tac-toe can be found in box (1.4) and is explained in-depth in this section.

1.3.1 Value arrays

Firstly an arbitrary value function V is needed. An initial value function equal to zero in all 3^9 states is created for this task. As described in the policy evaluation section, two arrays are needed for the old values (V_old) and the new values (V). After all states are evaluated, the values are saved in V_old, and new values will be estimated in V (line 6).

To estimate new values, the learning player goes over every possible state (line 9), *rounds* many times (line 5). The program iterates through all states from the terminal states (i.e., winning states for either player) to the starting state (i.e., empty board). A reward is received in the final states (X or O wins or a tie). Since each value is based on the previous value, estimates propagate quicker if we iterate from the terminal to the initial states of the game.

1.3.2 Board and state representation

A tic-tac-toe board filled with X's, O's, and empty squares is stored columnwise in a nine-element integer array. Mathematically the board S can be represented by $S := (s_i)_{i=1,\dots,9}$ where s_i are elements of $B := \{0, 1, 2\}$ which represents the X's (1), O's (2), and empty squares (0). In order to enumerate the state space instead of working with the nine-digit arrays which represent the board, two functions are used. The first function is *board2state* where a nine-digit array representing a board, is transformed into an integer (see box (4.2)):

$$board2state : \mathbb{Z}^9 \to \mathbb{Z} : S \to s = \sum_{i=1}^{9} s_i \cdot 3^{i-1}.$$

Similar there is a function *state2board* where state numbers are transformed into board arrays by using base-three (see box (4.1))

$$state2board : \mathbb{Z} \to \mathbb{Z}^9 : s \to b,$$

where $b := (b_{i,j})_{i,j=1,\dots,3} \in \mathbb{Z}^9$ and $b_{i,j} := S_{3(j-1)+(i-1)} \in B$.

1.3.3 Reachable states

Each state is established if this is a reachable state (line 11) and which players turn it is (line 14). Player X always starts, so if there are as many Xs as Os, the learning player X has his turn. If there is one X more than Os, player O has the turn (see code turn in box (4.3)). In all other cases, this is an unreachable state, and the variable *'reachableState'* is set to zero. If a player already won, it is also not intended to investigate the state any further, and *'reachableState'* is set to zero. A player wins when he has three marks in a row: diagonally, vertically, or horizontally, which is checked in box (4.5).

If the state is reachable, all admissible actions are established in line 16. Admissible actions are all empty squares of the board (indicated with a zero). The number of admissible actions is captured in *Nactions* (line 17). An action is represented as an array

$$A := \{a \cdot e_i | a \in B, i = 1, \dots, 9\},$$

where $\{e_i\}_{i=1,\cdots,9}$ denotes the canonical unit in $\mathbb{R}^9$. In this way, the state following the move is the vector sum of the state array S and the action array A.

If there is a tie, a player wins, or there are no actions available (lines 24 - 30), the value and policy are set to zero for this state and will be adjusted later on if needed. If none of these things are the case, the estimation of the values can start. Then, for each admissible action, the state and the action are given to the opponent to make a move.

1.3.4 Opponent

The function $p(s', x|s, a)$ is encompassed in the function 'opponent' in box (4.7). This function receives the current state s and action a from player X, a function for the opponent and (possibly) a policy π to be used by the opponent. The function defines an intermediate state *betweenstate*$= s+a$ and checks if the game is over, returning reward one if X has won and zero for a draw, and returning *new state* is -1 in this case. Otherwise, the intermediate state is passed to a subsequent function that determines s' and x. Once the subsequent function makes its move, there is checked if this led to winning, losing, or a draw, and the corresponding rewards are assigned.

The intermediate state is passed to an 'imperfect player in the first learning rounds.' which plays according to the following policy:

- If he can win (there are two Os in a row), he wins by playing the third O in the row.
- If he can block player X from winning (two Xs in a row), he does this by placing an O in this row.
- If there are no two Os or Xs in a row, he places an O in a (uniformly distributed) random empty square (this is why this player is 'imperfect')

The imperfect player prioritizes in the following order: winning, blocking, and finally a random move (see box (4.8)). The critical state function in box (4.9) sets the variables to check if it is possible to win or block.

After a few iterations, the learning player will improve his policy to reach an optimum. This optimum will not win 100% of the time since the imperfect player often plays a random move. However, winning from an imperfect player is not the biggest challenge to overcome. Therefore, to approach an optimal policy and beat more demanding players, the learning player can play against itself to further improve his policy. In this way, the programmer does not have to program better players every time the learning player is done learning, and more importantly, by playing against itself, the human intellect is held out of the loop. In box (4.10) is seen how the opponent uses the current policy *Pi* to make moves. The function receives a state and the current policy. Firstly the turn of the player is determined. Secondly, the probability of making each action according to the policy is searched. Whenever the probability of choosing an action is not zero, the action is tried, and the new state is given (line 13).

Figure 1.8: Tic-tac-toe: state 11457

Example 1.5. *For example if the current state is 11457, the function state2board gives the board representation:* $B = [1, 2, 0, 2, 0, 1, 1, 0, 0]$ *(figure (1.8)). The player who's turn it is is player O, so turn(state)* $= 2$*. For the imperfect player there is no chance to win or to block, so he can make four random moves at the thirth, fifth, eighth and ninth place. Hence imperfect player(state)* $= [12915, 11619, 11463, 11459]$ *which correspond to the following boards:*

$$[1, 2, \boldsymbol{2}, 2, 0, 1, 1, 0, 0]$$
$$[1, 2, 0, 2, \boldsymbol{2}, 1, 1, 0, 0]$$
$$[1, 2, 0, 2, 0, 1, 1, \boldsymbol{2}, 0]$$
$$[1, 2, 0, 2, 0, 1, 1, 0, \boldsymbol{2}]$$

The policy $\pi(s)$ *in state 11457 is:* $[0, 0, \frac{1}{4}, 0, \frac{1}{4}, 0, 0, \frac{1}{4}, \frac{1}{4}]$.

*Next: let us give state 11457 to an opponent with a policy that has been improved already (policy Pi*1*, which will be discussed later on). It turns out that making a move in the fifth and eighth place has a higher probability of winning. Indeed the fifth place will create a possibility of winning horizontally and prevent X from winning diagonally. The eighth-place creates the possibility to win horizontally and prevents player X from getting a vertical row. The third and ninth place only prevents player X from winning but does not provide a possibility of winning itself. Hence* $Pi1(11457) = [0, 0, 0, 0, \frac{1}{2}, 0, 0, \frac{1}{2}, 0]$.

1.3.5 Value update

For each of the admissible actions the value is calculated. If the action brings the state in a new state (=-1) where a player wins or there is draw, the value of this state is set to the corresponding reward (line 42). In the other cases the value is calculated in the following way:

$$\text{Vprime(action)} \leftarrow \frac{\text{reward} + \gamma \cdot \text{V\_old(new state)}}{\text{Nactions}} \tag{1.19}$$

The action(s) which gives the maximum value is captured and is called *a_max* and the maximum value: *V_max*, which is saved for the state. Notice how in box (1.3) the value function is updated in the following way:

$$V(s) \leftarrow \max_a \sum_{s',x} p(s', x|s, a)[x + \gamma V(s')] \tag{1.20}$$

In the case of tic-tac-toe the probability $p(s', x|s, a)$ is encompassed in the function 'opponent'. Hence (1.19) and lines 50 and 51 combine to (1.20). There can be more then one action leading to the same maximum value. The action(s) leading to the highest value are saved in a_{max}. Next $\pi_t(a|s)$ is set to the probability $\frac{1}{|a_{max}|}$ and the other actions are set to zero (see example (1.5)).

1.3.6 Results: imperfect player

The win, lose and tie rate according to a policy against another policy is determined by the algorithm in box (4.11), where the game is played *numGames* times. To fairly determine how good one policy against another is: the first player is picked at random in line 10. Then, the players play alternately and make a move based on the current state and its policy function until a win or a draw is detected.

The policy and value function are set to zero in the first round. Then this policy is trained against the imperfect player for 1000 (='numGames') games, resulting in policy 1.

Policy 1 wins 67,8%, loses 1,5% and ties 30,7%. These results are obtained by the *testPolicy* function where *Pi1* is policy 1, *Pi2* is the all zeros policy and *OPPfun* is the imperfect player.

If policy 1 is used to train against the imperfect player again for 1000 rounds, policy 2 is obtained. Policy 2 wins 69,2%, loses 10%, and ties 29,8%. If policy 2 is trained against the imperfect player, resulting in policy 3, and policy 3 resulting in policy 4, etcetera, the results are shown in table (1.1). Since the winning rate is not significantly increasing, it is decided to continue with policy 1 to learn from itself.

1.3.7 Results: playing against itself

In the previous paragraph, a value function and policy are found by playing against an imperfect player with a minimum loss rate of 10,0%. In this section, the learning player improves its policy by playing against itself and the previous policy. Since the loss rate is not significantly decreasing when training against an imperfect player (table (1.1)), it is chosen to continue with policy 1 to learn from itself. Thus policy A is created by training against policy 1, i.e., using in the value iteration algorithm: the values of policy 1, policy 1 and the *OPPfun* is playPolicy.

player one vs	**player two**	**win**	**lose**	**draw**
policy 1	imperfect player	678	15	307
policy 2	imperfect player	692	10	298
policy 3	imperfect player	694	13	293
policy 4	imperfect player	694	19	287
policy 5	imperfect player	692	20	288

Table 1.1: Results of training against imperfect player

The algorithm for playPolicy is given in box (4.10). The algorithm checks whose turn it is (line 3), gives the current state to the policy (line 7), which returns the best move according to that policy (line 12). The algorithm makes a move (line 13) and returns the new state. If policy 2 is tested against the imperfect player, the win rate goes down compared to policy 1, but it has a 0% loss rate! If policy A then is tested against policy 1 it also never loses, whereas policy 1 against policy 1 loses 48% of the time. Hence policy A is an improvement in comparison with policy 1. Finally, policy A is tested against itself, and this ends only in draws; no wins or losses were recorded.

player one vs	**player two**	**win**	**lose**	**draw**
policy 1	imperfect player	678	15	307
policy A	imperfect player	466	0	534
policy A	policy 1	283	0	717
policy 1	policy 1	520	480	0
policy A	policy A	0	0	1000

Table 1.2: Results of training against imperfect player and its own policy.

```
function [V,Pi] = value_iter_all(V,Pi,rounds,OPPfun)

gamma=0.9;

for iter = 1:rounds,
    V_old = V;
    Pi_old = Pi;

    for state = [3^9-1:-1:1,3^9]

        if reachableState(state),

            B = state2board(state);
            player = turn(state);

            action_list = find(B==0);
            Nactions = length(action_list);

            %, if it is 'my turn' and the state is 'reachable', then:p1
            % (1) the other player has just one
            % (2) the game is a draw
            % (3) I have not yet won and can make another move

            if checkWin(state,3-player)
                V(state) = 0;
                Pi(state,action_list) = 0;

            elseif (Nactions==0)
                V(state) = 0;
                Pi(state,action_list) = 0;

            else
                Vprime = zeros(1,Nactions);

                for a = 1:Nactions,
                    action = zeros(1,9);
                    action(action_list(a)) = player;

                    [newstate,reward] = ...
                        opponent(state,action,OPPfun,Pi_old,false);

                    if newstate == -1
                        Vprime(a) = reward; %(1+gamma)*reward;

                    else
                        Vprime(a) = mean(reward + gamma * V_old(newstate));

                    end
                end

                Vmax = max(Vprime);
                amax = find(Vprime==Vmax);

                V(state) = Vmax;
                Pi(state,:) = 0;
                Pi(state,action_list(amax)) = 1/length(amax);
            end
        end
    end
end
end
```

Box 1.4: Value iteration algorithm: tic-tac-toe

2 Surrogate reward function based on graph Laplacians

In the previous chapter, the basics of reinforcement learning algorithms are explained. The tic-tac-toe game is used as an example, which has a straightforward definition for its reward function (1.1) and is part of the problem definition. Winning gives a reward of one, losing minus one, and in all other cases, the reward is zero. Unfortunately, such an easily defined reward function may provide little information in the early stages of (episodic) learning, when the state is far from the goal. This chapter investigates how a surrogate reward function can be defined, using the graph Laplacian, to speed up learning.

As a motivating example through this chapter, a walking game is considered. The walker (agent) walks through a grid world of 20 by 20 cells (environment) and wants to reach the terminal cell in the fastest way (goal). A horizontal wall is built from (0,10) until (15,10) to inhibit the walker. The walker cannot go through the wall but beforehand does not know where the wall is. The walker can move in four directions: north, south, west, and east (actions), one step at a time. According to Nachum et al. (2018), an obvious choice for the reward function is the negative Euclidean distance from each cell to the terminal cell. How the Euclidean distance is computed depends on the representation of the states. If the states are represented tabularly, the reward usually used is the sparse reward. The sparse reward gives the same negative value for all non-terminal cells and zeroes for the terminal cell. However, the tabular representation can be disadvantageous for the learning speed (Wu et al., 2018). Instead, one may use the grid cells' (x, y) representation to compute the Euclidean distance. When moving closer to the terminal cell, the agent receives a higher reward, increasing the agent's learning speed compared to the sparse reward (Wu et al., 2018).

In figure (2.1), the reward structure is seen when the terminal cell is (3,6). The Euclidean reward function is not able to see the wall. When a reinforcement learning algorithm uses the Euclidean distance as a surrogate reward function, it will likely converge to a policy that will go through the wall. For example, if the walker starts at (3,11), the algorithm

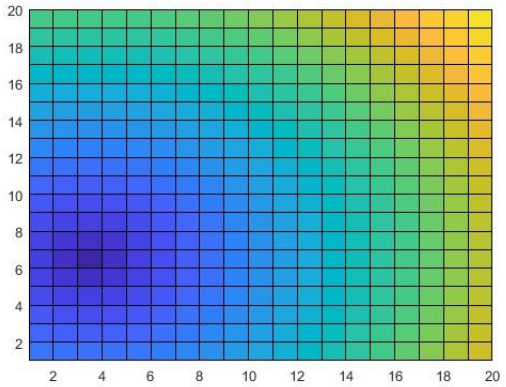


Figure 2.1: Euclidean distance on (x,y) representation.

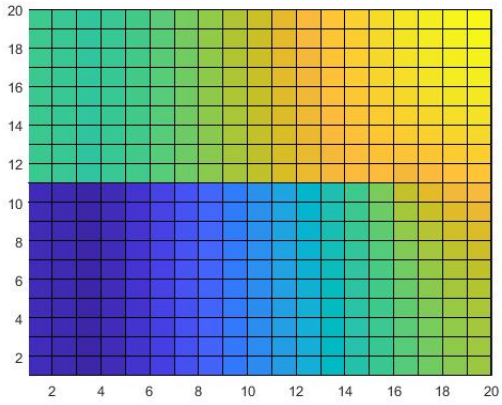


Figure 2.2: Second eigenvector of the graph Laplacian: showing awereness of the wall in the walking game.

will probably give a policy that goes from (3,11) only south to reach the terminal cell in five steps. However, it is impossible to travel from (3,10) to (3,9) since the wall blocks this step. The surrogate reward function should tell the model if a state can reach another state closer or further away from the terminal cell to avoid convergence to suboptimal policies like this.

It turns out that the eigenvectors of the graph Laplacian are very useful in several situations like graph visualization (Yehuda, 2003), clustering (Andrew et al., 2002), and more (Chung et al. 1997). Further, Mahadevan (2005) uses the eigenvectors of the graph Laplacian as geodesically smooth orthonormal basis functions for approximating value functions, increasing learning speed. Also, Wu et al. (2018) suggest using the d smallest eigenvectors of the Laplacian for a surrogate reward function in the walking game. Wu et al. (2018) use the uniformly random behavior policy (i.e., going to all directions with equal probability). This leads to a walker conscious of the walls in their grid world and induces a distance function. Elaborating on this, Wu et al. (2018) show that this choice of representing avoids suboptimal policies (unlike the Euclidean reward function) and increases the learning speed. In this chapter, several graph Laplacians are defined and used to define a surrogate reward function for the walking game, with a promising result as shown in figure (2.2).

2.1 Definitions of the graph Laplacian

As perhaps noticed, the *graph* Laplacian is calculated from a graph. Hence, some basic notions of graph theory are given in section (2.1.1), followed by definitions of graph Laplacians. Most games can be transformed into a graph. For example, the walking game can be transformed into a graph by defining the states as vertices of the graph, and the edges are from each state to its four neighbors in the directions: north, south, west, and east (see figure (2.3)). In addition, the edges are weighted by the transition probability from going from one state to the other. Whereby edges that go through the wall or off the grid are given zero weight. Likewise, the tic tac toe game can be transformed into a graph. Each vertex represents a state, and edges are drawn between two consecutive moves from X and O.

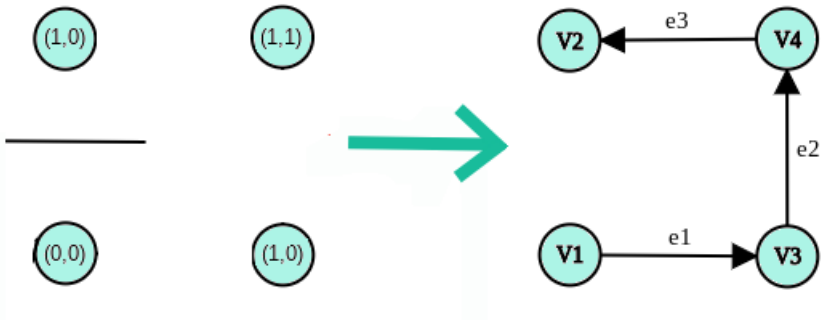


Figure 2.3: Left: (x,y) representation of a four cell walking game.
Right: directed graph: G_1.

2.1.1 Graph theory

First, the definitions for directed and undirected graphs are given, each followed by the definition of the graph Laplacian. Secondly, weighted graphs are defined, and the specific transition probability matrix for the walking game is given.

Directed graphs

Definition 2.1. A **directed graph** is a pair $G = (V, E)$ where $V = \{v_1, \ldots, v_m\}$ is a set of **vertices** and $E \subseteq V \times V$ is a set of ordered pairs of distinct vertices ($(u, v) \in V \times V$ with $u \neq v$), called **edges** (Gallier and Quaintance, 2020).

As an example consider a four cell walking game as visualized on the left side of figure (2.3), with a small horizontal wall between (0,0) and (1,0). Transforming this to a directed graph gives graph G_1 on the right side of figure (2.3). Each coordinate is transformed into a vertex $(v_1, \ldots, v_4)$ and all possible edges are drawn (e_1, e_2, e_3). There is no edge between v_1 and v_2 since there is a wall between (0,0) and (1,0).

Definition 2.2. For every node $v \in V$ the **degree** $d(v)$ of v is the number of edges escaping or arriving v

$$d(v) := |\{u \in V | (u, v) \in E \text{ or } (v, u) \in E\}|.$$

The corresponding **degree matrix** $D(G)$ is the diagonal matrix $D(G) = \text{diag}(d_1, \ldots, d_m)$ (Gallier and Quaintance, 2020).

Since v_3 and v_4 are connected with two edges and the others with one edge, the matrix below is the degree matrix of graph G_1:

$$D(G_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}. \tag{2.1}$$

Definition 2.3. Given a directed graph G the **incidence matrix** $B(G)$ of G is the $m \times n$ matrix whose entries $b_{i,j}$ are given by (Gallier and Quaintance, 2020):

$$b_{i,j} := \begin{cases} +1 & \text{if } s(e_j) = v_i \\ -1 & \text{if } t(e_j) = v_i \\ 0 & \text{otherwise,} \end{cases}$$

where s is the source vertex and t the target vertex.

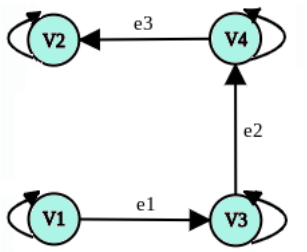


Figure 2.4: Directed graph with selfloops: G_2.

So for the directed graph G_1, the incidence matrix is the following:

$$B(G_1) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}. \tag{2.2}$$

Definition 2.4. (Gallier and Quaintance, 2020) Given a directed graph $G = (V, E)$, with $V = \{v_1, \ldots, v_m\}$ the **adjacency matrix** $A(G)$ of G is the symmetric $m \times m$ matrix $(a_{i,j})$ such that:

$$a_{i,j} := \begin{cases} 1 & \text{if there is some edge}(v_i, v_j) \in E \text{ or } (v_j, v_i) \in E \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix $A(G_1)$ is:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}. \tag{2.3}$$

Theorem 2.1. *Given any directed graph G if B is the incidence matrix, A the adjacency matrix, and D is the degree matrix of G, then*

$$L = BB^T = D - A.$$

Hence BB^T is independent of the orientation of G. Also, $L = D - A$ is positive, semidefinite, symmetric (i.e., the eigenvalues of L are nonnegative) (Gallier, 2013).

Hence the long-awaited graph Laplacian for directed graphs can be defined.

Definition 2.5. The matrix $L := BB^T = D - A$ is defined as the **graph Laplacian** of the directed graph G, where B is the incidence matrix of G, A is the adjacency matrix of G, and D is the degree matrix of G (Gallier, 2013).

The graph Laplacian of graph G_1 is the following:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} - \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{pmatrix}.$$

Notice how self-loops are excluded in definition (2.1), since edges are pairs (u, v) with $u \neq v$ and there is at most one edge from a vertex u to a vertex v. If self-loops are allowed for each vertex, the degree of each vertex goes up by one (i.e. $d(v) = d(v) + 1$ hence $D(G_2) = D(G_1) + I$) also the adjacency matrix gets an extra one at the diagonal for each self-loop (i.e. $A(G_2) = A(G_1) + I$). Hence $D(G_1) - A(G_1) = D(G_2) - A(G_2)$ are the same if each vertex of G_1 has one self-loop. Hence the graph Laplacian of G_1 is the same as the the graph Laplacian of G_2:

$$L(G_1) = D(G_1) - A(G_1) = D(G_2) - A(G_2) = L(G_2). \tag{2.4}$$

This result is handy because later, it will be clear that the walking game is a directed graph with self-loops if the probability of going in each direction is not uniform.

Undirected graphs

Undirected graphs are realized by removing the orientation of directed graphs. In figure (2.5) is the undirected graph G_3 obtained from the directed graph G_1 in figure (2.3) seen.

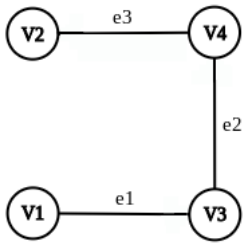


Figure 2.5: Undirected graph: G_3.

Definition 2.6. An **undirected graph** is a pair $G = (V, E)$, where $V = \{v_1, \ldots, v_m\}$ is the set of vertices, and E is the set edges which consist of two-element subsets of V (that is, subsets $\{u, v\}$, with $u, v \in V$ and $u \neq v$) (Gallier and Quaintance, 2020).

Definition 2.7. For every node $v \in V$, the **degree** $d(v)$ of v is the number of edges escaping or arriving v:

$$d(v) := |\{u \in V | (u, v) \in E\}|.$$

The **degree matrix** D is defined as in the directed graph case (definition (2.2)) (Gallier and Quaintance, 2020).

The degree matrix of directed and undirected graphs is equal, hence $D(G_3)$ can be found in equation (2.1) and $D(G_1) = D(G_3)$.

Definition 2.8. Given an undirected graph G the **incidence matrix** $B(G)$ of G is the $m \times n$ matrix whose entries $b_{i,j}$ are given by (Gallier and Quaintance, 2020):

$$b_{i,j} := \begin{cases} +1 & \text{if } e_j = \{v_i, v_k\} \text{ for some } k \\ 0 & \text{otherwise.} \end{cases}$$

Hence the incidence matrix of the graph G_3 is:

$$B(G_3) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}. \tag{2.5}$$

Notice how the entries of the incidence matrix are nonnegative for an undirected graph, but for directed graphs, the incidence matrix also contains negative values (see (2.2)).

Definition 2.9. Given an undirected graph $G = (V, E)$, with $V = \{v_1, \ldots, v_m\}$ the **adjacency matrix** $A(G)$ of G is the symmetric $m \times m$ matrix $(a_{i,j})$ such that (Gallier and Quaintance, 2020):

$$a_{i,j} := \begin{cases} 1 & \text{if there is some edge}(v_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

The adjacency matrix for G_3 is the same as for G_1: $A(G_1) = A(G_3)$ given in equation (2.3). Next the graph Laplacian for undirected graphs is defined.

Definition 2.10. The matrix $L := D - A$ is defined as the **graph Laplacian** of the undirected graph G, where B is the incidence matrix of G, A is the adjacency matrix of G, and D is the degree matrix of G (Gallier and Quaintance, 2020).

The graph Laplacian of the undirected graph G_3 is equal to the graph Laplacian of the directed graph G_1 since:

$$L(G_1) = D(G_1) - A(G_1) = D(G_3) - A(G_3) = L(G_3). \tag{2.6}$$

Equation (2.6) is also, in general, true if the undirected graph is deduced from the directed graph by removing the orientation of the edges. Combining equation (2.6) with equation (2.4) yields undirected graphs with self-loops that have the same Laplacian as the undirected graph without self-loops. This is convenient because the walking game with uniform direction probability is an undirected graph with self-loops.

Weighted graphs

In typical reinforcement learning problems, one wants to find the optimal policy by finding the optimal value function. For example, suppose the problem is transformed into a graph and the direction probability is uniform. In that case, the optimal policy is the shortest path (least number of steps) from the initial state to the terminal state. However, when thinking of the walking game, the shortest path might not always be a feasible solution if the path, for example, passes the wall. Furthermore, if the walk is biased, the shortest path might not be optimal (an optimal policy wants to take advantage of the bias). So not in all problems, the shortest path represents the optimal policy. Hence weights are added to the edges to exclude infeasible solutions and include environmental factors like a north-east wind. First, the definitions of a general weighted graph, its degree matrix, and the Laplacian are given. Secondly, the transition probability matrix will be explained, used as a weight matrix for the walking game.

Definition 2.11. A **weighted graph** is a pair $G = (V, W)$ where $V = \{v_1, \ldots, v_m\}$ is the set of vertices and W is the weight matrix, such that $w_{i,j} \geq 0$ for all $i, j \in \{1, \ldots, m\}$. The corresponding (undirected) graph (V, E) with $E = \{\{v_i, v_j\} | w_{i,j} > 0\}$ is called the underlying graph of G (Gallier and Quaintance, 2020).

For weighted graphs, the degree matrix is dependent on the weights, as seen in the following definition.

Definition 2.12. For every node $v_i \in V$ the **degree** $d(v_i)$ of v_i is the sum of the weights of the edges adjacent to v_i:

$$d(v_i) := \sum_{j=1}^{m} w_{i,j}.$$

The corresponding **degree matrix** $D(G)$ is defined as before: $D(G) = \text{diag}(d_1, \ldots, d_m)$ (Gallier and Quaintance, 2020).

The weighted matrix of a graph and the degree matrix give rise to a new definition of a graph Laplacian.

Definition 2.13. (Gallier, 2013) Given any weighted graph $G = (V, W)$ with $V = \{v_1, \ldots, v_m\}$, the **graph Laplacian** $L(G)$ of G is defined by:

$$L(G) := D(G) - W.$$

In theorem (2.1) is seen that the Laplacian of directed graphs can be calculated by the degree matrix and the adjacency matrix, but also by the incidence matrix and its transpose. For weighted graphs, something similar holds, but not all (weighted) graphs are directed. If an undirected weighted graph is considered, the orientation of the graph is defined according to the following definition.

Definition 2.14. Given any undirected graph $G = (V, E)$, an orientation of G is a function $\sigma : E \to V \times V$ assigning a start and an end to every edge in E, which means that for every edge $\{u, v\} \in E$, either $\sigma(\{u, v\}) = (u, v)$ or $\sigma(\{u, v\}) = (v, u)$. The **oriented graph** G^σ obtained from G by applying the orientation σ is the directed graph $G^\sigma = (V, E)$, with $E^\sigma = \sigma(E)$ (Gallier, 2013).

This gives all the tools to define the incidence matrix for weighted graphs.

Definition 2.15. Given a weighted graph $G = (V, W)$, with $V = \{v_1, \ldots, v_m\}$, if $\{e_1, \ldots, e_n\}$ are the edges of the underlying graph of G for any oriented graph G^σ obtained by giving an orientation to the underlying graph of G, the **incidence matrix** B^σ of G^σ is the $m \times n$ matrix whose entries $b_{i,j}$ are given by (Gallier and Quaintance, 2020):

$$b_{i,j} := \begin{cases} +\sqrt{w_{i,j}} & \text{if } s(e_j) = v_i \\ -\sqrt{w_{i,j}} & \text{if } t(e_j) = v_i \\ 0 & \text{otherwise.} \end{cases}$$

Gallier (2013) found that the Laplacian of weighted graphs (definition (2.13)) can be calculated by not only the degree matrix and the weight matrix, but also by the oriented graph and its transpose, as can be seen in the following theorem.

Theorem 2.2. *Given any weighted graph $G = (V, W)$ where V is the set of vertices $\{v_1, \ldots, v_m\}$, W is the weight matrix and D the degree matrix of G. If B^σ is the incidence matrix of the oriented graph G^σ, then*

$$L = B^\sigma (B^\sigma)^T = D - W.$$

Hence $B^\sigma (B^\sigma)^T$ is independent of the orientation of G. Also, $L = D - W$ is positive, semidefinite, symmetric (i.e., the eigenvalues of L are nonnegative) (Gallier, 2013).

If every row of the weight matrix W contains at least one positive entry the graph G has no isolated vertices. On this assumption all elements of the degree matrix D, as defined in (2.12), contain positive entries and hence is invertible. So $D^{-\frac{1}{2}} := \text{diag}(d_1^{-\frac{1}{2}}, \ldots, d_m^{-\frac{1}{2}})$ makes sense and two more Laplacians can be defined (Gallier, 2013).

Definition 2.16. Given any weighted directed graph $G = (V, W)$ with no isolated vertex and with $V = \{v_1, \ldots, v_m\}$, the graph Laplacians L_{sym} and L_{rw} of G are defined by:

$$L_{sym} := D^{-\frac{1}{2}} \cdot L \cdot D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} \cdot W \cdot D^{-\frac{1}{2}}$$
$$L_{rw} := D^{-1} \cdot L = I - D^{-1} \cdot W.$$

Remark that L_{sym} is a symmetric matrix since both L and $D^{-\frac{1}{2}}$ are symmetric matrices. In addition we have:

$$\begin{aligned} L_{rw} &= D^{-1} \cdot L \\ &= D^{-\frac{1}{2}} \cdot D^{-\frac{1}{2}} \cdot L \cdot I \\ &= D^{-\frac{1}{2}} \cdot (D^{-\frac{1}{2}} \cdot L \cdot D^{-\frac{1}{2}}) \cdot D^{\frac{1}{2}} \\ &= D^{-\frac{1}{2}} \cdot L_{sym} \cdot D^{\frac{1}{2}}. \end{aligned}$$

The Laplacian L_{rw} is associated with the random walk on the graph G (Gallier, 2013).

2.1.2 Transition Probability Matrix walking game

In this example, the walking game is played on a $K \times M$ grid, $K = M = 20$. Each vertex not adjacent to a wall has an edge in the directions: north, south, west, and east to another vertex. The transition probability matrix gives a weight to each edge, whereby a distinction is made between the types of edges. The algorithm for the transition probability matrix is given in box (5.3) and is clarified in this section. In principle, each direction is given weight $\frac{1}{4}$ such that the edges of each vertex sum up to one, see figure (2.6). An

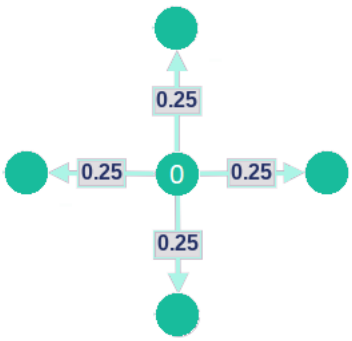


Figure 2.6: Vertex zero and its weights.

exception is a corner vertex, where only two directions of movement are possible. These two directions are also given weight $\frac{1}{4}$ but in addition the self-loop to this vertex is given weight $\frac{1}{2}$ to still sum up to one (see figure (2.7)). Another exception is the vertex on the border of the grid. The border vertex has three directions to go, and hence the self-loop is given weight $\frac{1}{4}$ (see figure (2.8)).

The last special vertex is one step away from the wall. The wall is defined on the $K \times M$ grid from $i = 1, \ldots, \frac{M}{2} + 1$ and $j = 1, \ldots, 3 \cdot \frac{K}{4}$.Hence the lines $i = 10$ and $i = 11$ both from $j = 1, \ldots, 15$ are on both sides of the wall of this grid. The weights can be seen in figure (2.9).

If there is a strong bias, for example, the probability of going in the directions north and east is higher than west and south. See for example the following direction probability vector:

$$p = [\text{north} = 0.28, \text{east} = 0.35, \text{south} = 0.22, \text{west} = 0.15]$$

This vector results in the weights per direction as shown in figure (2.10). This also affects the weights going from a corner or wall vertices as shown in figures (2.11) and (2.12).

The transition matrix P (as defined in box (5.3)) will be used as a weight matrix for the walking game to compute the Laplacian. As seen above, self-loops are needed; this

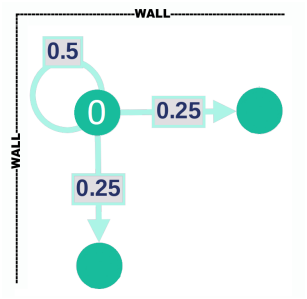


Figure 2.7: Corner-vertex zero and its weights.

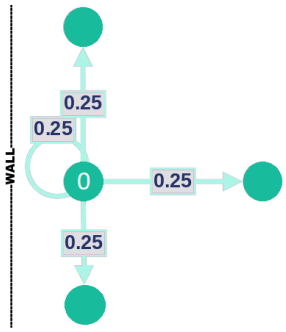


Figure 2.8: Border-vertex zero and its weights.

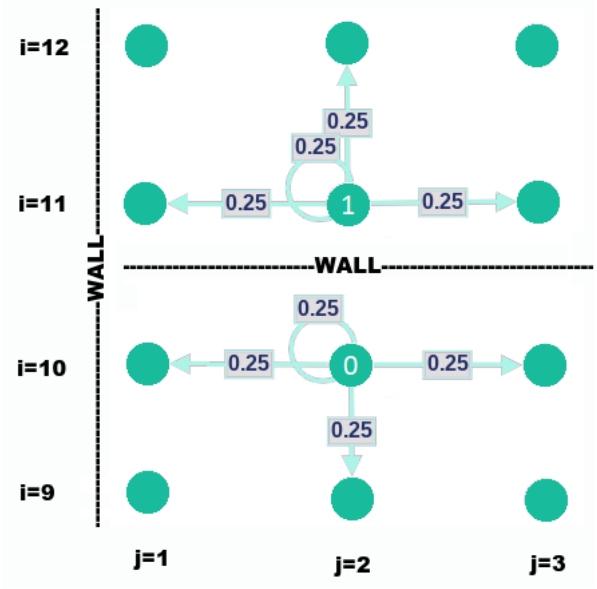


Figure 2.9: Vertices zero and one and their weights.

might be problematic since the graph Laplacian for weighted graphs is only defined for weight matrices without self-loops (definition (2.13)). In theorem (2.2) is seen that the Laplacian is positive, semidefinite and symmetric. The transition matrix $P = (w_{ij})$, is a 400×400 symmetric matrix. If $L = D - W$ and D is the degree matrix as defined in definition (2.12) then for all $x \in R^{400}$ holds:

$$x^T L x = \frac{1}{2} \sum_{i,j=1}^{400} w_{i,j}(x_i - x_j)^2. \tag{2.7}$$

Hence $x^T L x$ is independent of $w_{i,i}$ for all i (i.e., all selfloops). Also, if $w_{i,j} \geq 0$ for all $i, j \in \{1, \ldots, 400\}$ the Laplacian L is still positive semidefinite (Gallier, 2013) and hence self-loops are allowed for computing the Laplacian.

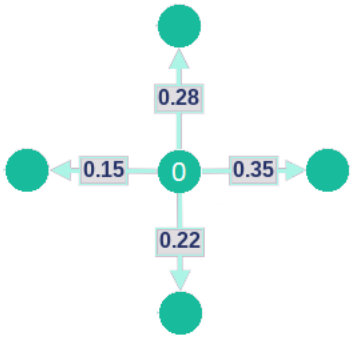


Figure 2.10: Vertex zero and its weights with a north-east wind.

Now that the transition probability matrix is defined, a final Laplacian can be defined. The transition probability is a Markov chain and is assumed to have a unique stationary distribution ρ such that for any measurable $U \subset \mathcal{S}$ holds $\rho(U) = \int_{\mathcal{S}} P^{\pi}(U|v)d\rho(v)$ (Wu et al., 2018).

Definition 2.17. (Wu et al., 2018) If the state set $\mathcal{S}$ is finite, ρ is the stationary distribution of the transition distribution P^{π} with respect to a fixed policy π then the pairwise affinity between two vertices u and v on the graph is

$$L_{wu} := \frac{P^{\pi}(s_{t+1} = v|s_t = u)}{\rho(v)} + \frac{P^{\pi}(s_{t+1} = u|s_t = v)}{\rho(u)}.$$

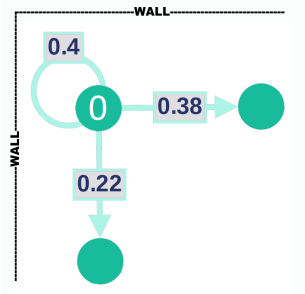


Figure 2.11: Corner-vertex zero and its weights with a north-east wind.

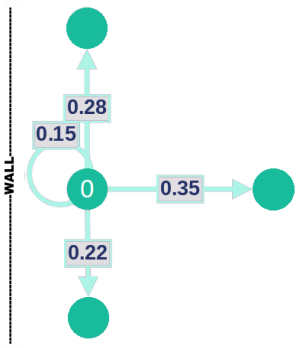


Figure 2.12: Border-vertex zero and its weights with a north-east wind.

2.2 Value iteration algorithm for the walking game

In the previous section, a few Laplacians are defined. The second smallest eigenvalue λ_2 of the Laplacian matrix, often mentioned as the **Fiedler number**, holds much information about the graph. Fiedler himself calls λ_2 the algebraic connectivity of the graph G: *"If the eigenvalue is greater than zero, the graph is connected."* - Fiedler, (1998). This section explains how a value iteration algorithm for the walking game is programmed and how the second eigenvector (i.e., **Fiedler vector**) of the Laplacian can be used as a surrogate reward function.

2.2.1 Sparse value function

Before defining the surrogate reward function based on the Laplacian, the value function for the sparse reward is described in this section. Wu et al. (2018) claim that when the Laplacian reward function is used to calculate the value function, its learning speed increases compared to that of the value function learned when using the sparse reward (see below). Therefore, the learning speed will be compared for the walking game, and the sparse value function is needed.

Like in the previous chapter, at time t, the agent is in a state $s_t \in \mathcal{S}$ (i.e., cell in the grid). There is a policy $\pi(a|s_t)$ which is the probability of choosing an action (i.e., north, south, west, or east) at time t for a given state s_t. After making an action, the environment provides a new state s_t and a reward r_t from the reward distribution function $R(s_t, a_t)$, which is the main goal of defining in this chapter. The probability of going in a certain direction is chosen beforehand. At first, each direction has the same probability (i.e., $p_N = p_S = p_W = p_E = \frac{1}{4}$) and the corresponding policy π_u is called **the uniform policy** (or sometimes **random walk policy**), where

$$\pi_u(a|s) := \begin{cases} 0 & \text{if } a \text{ causes to hit a wall} \\ \frac{1}{4} & \text{otherwise.} \end{cases} \tag{2.8}$$

Later other probabilities will be considered too.

As mentioned in the first section of this chapter, the reward usually used if states are represented tabularly is the sparse reward. **The sparse reward** function gives the following rewards:

$$R_s := \begin{cases} 1 & \text{if } s \text{ is the terminal state} \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

The sparse value function (V_s) has the same structure as explained in section (1.2.6). Each iteration starts in the start state (given by the user) and ends when the terminal state is reached (also given by the user). The uniform policy is used to go to the next state in each state. The value is calculated in the following way:

$$V(s) = V(s) + R_s(s') + \gamma \cdot V(s') \text{ where: } \begin{cases} s = \text{old state} \\ s' = \text{new state.} \end{cases}$$

The value iteration needs approximately 15 iterations (called *episodes*), and each iteration needs to explore $\sim 4.98 \cdot 10^4$ states (called *steps*) before terminating when the uniform policy is used without adaptive learning. The algorithm terminates when the finish state is reached. When the algorithm does use *adaptive learning*, the mean number of episodes

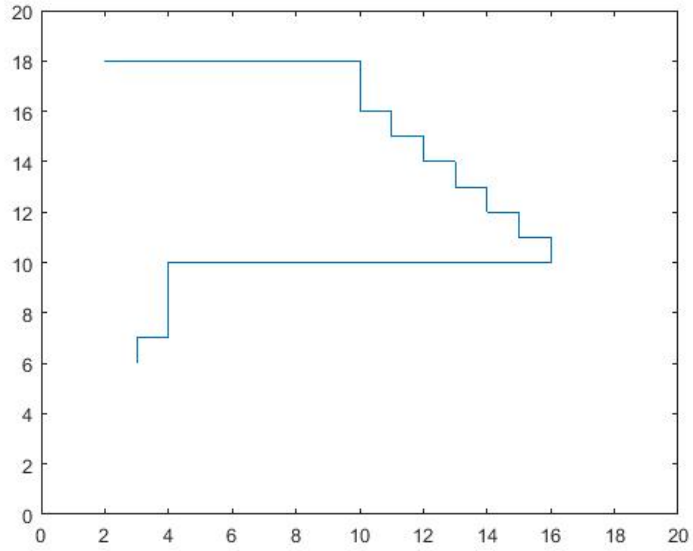


Figure 2.13: Minimum path length from $[2, 18]$ to $[3, 6]$ when using V_{exact}.

is 14.80, and the mean number of steps decreased slightly to $\sim 4.12 \cdot 10^4$. *Adaptive learning* means that the algorithm starts with using the uniform policy and, after learning some values, uses **the deterministic policy**, π_d in μ percentage of the time (i.e., the best move in the state based on the current learned values):

$$\pi_d(a|s) := \begin{cases} 1 & \text{if } a \text{ leads to a neighbour state of } s \text{ with maximum value.} \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

The algorithm can be found in box (5.5) and can be used by box (5.7) and (5.8) by uncommenting the uniform policy (line 41) and commenting the mixed policy (line 42). The adaptive policy is used when lines 31-33 are uncommented.

In reinforcement learning, one typically does not want to search the whole state space to find a value function (see section (2.4)). Instead, reinforcement learning plays the game repeatedly and tries to learn from local information. Since for this specific walking game, the dynamics are completely known, and the number of states (400) is small, the value function can be calculated exactly as described in equation (1.11):

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',x} p(s', x|s, a)\big[x + \gamma v_\pi(s')\big].$$

This value function uses the sparse reward (i.e. $x \in R_s$) and will be referred to as V_{exact} := **the exact sparse value function**. The algorithm can be found in box (5.4). The Bellman equation is a recursive relation since the only non-zero reward received in the sparse reward function is at the terminal state. The terminal state is the starting point of the algorithm. The algorithm continues by visiting the direct neighbors in west, south, east, and north directions and calculating the values in these states, continuing by visiting the neighbors' neighbors, etc. The minimum length path from start state $[2, 18]$ to terminal state $[3, 6]$ has length 40 and is nonunique. One optimum is shown in figure (2.13).

The sparse value function converges to the exact value function if the random walker takes approximately $\sim 10^7$ steps. On the y-axis of figure (2.14) the error between V_s and V_{exact}, calculated by $\|V_s - V_{exact}\|$ is seen. On the x-axis, the number of steps of the

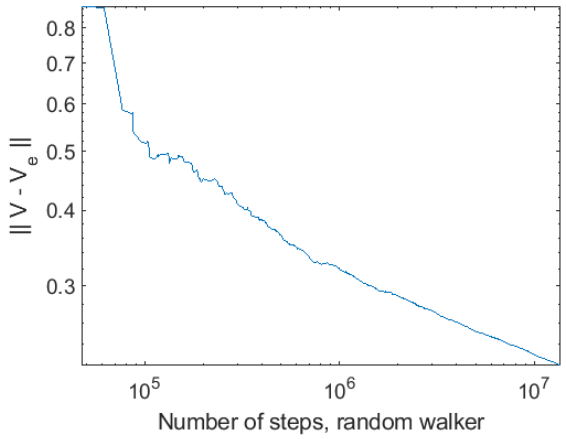


Figure 2.14: Difference between the sparse value function and the exact sparse value function against the number of steps of the random walker.

random walker is shown. The algorithm for the calculation of the converged value function can be found in box (5.6).

2.2.2 Laplacian value function

To compute the value function with a reward based on the second value of the eigenvector of the Laplacian, the same value iteration algorithm is used as for the sparse reward function (box (5.5)). The obvious difference is the reward function used. The first section of this chapter defines the Laplacian for directed, undirected, and weighted graphs. The algorithm for the transition probability matrix for the walking game is given in the second section. Then, the transition probability matrix P is used as the weight matrix of the graph. Finally, the four Laplacians for weighted graphs are given in the following overview and are calculated in box (5.3) lines 27-59:

- $L = D - P$: definition (2.13).
- $L_{sym} = D^{-\frac{1}{2}} \cdot L \cdot D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} \cdot P \cdot D^{-\frac{1}{2}}$: definition (2.16).
- $L_{rw} = D^{-\frac{1}{2}} \cdot L_{sym} \cdot D^{\frac{1}{2}}$: definition (2.16).
- $L_{wu} = \frac{P^{\pi}(s_{t+1}=v|s_t=u)}{\rho(v)} + \frac{P^{\pi}(s_{t+1}=u|s_t=v)}{\rho(u)}$: definition (2.17).

The reward function for each Laplacian is calculated in the same way. First the second eigenvector of the Laplacian is calculated (X_2), then the distance from the eigenvector to the terminal state (with respect to the eigenvector) is calculated by:

$$R_2(s) := \sqrt{(X_2(s) - X_2(\text{terminal state}))^2}.$$

Finally, the reward function for the Laplacian is defined by:

$$R_L(s) := 1 - \frac{R_2(s)}{\max_s R_2(s)}. \tag{2.11}$$

As seen in chapter one, the proportion between exploration and exploitation in reinforcement learning needs to be made. When the walker exploits, it only chooses actions it has already made before and uses this knowledge to maximize the total reward. However, exploring new areas of the environment is also necessary (Douglas, 2021). Let $1 - \mu$

	Value function	Policy	Reward
V_s	Sparse value function	π_u	R_s
V_{exact}	Exact sparse value function	π_u	R_s
V_m	Mixed Laplacian value function	π_m	R_m

Table 2.1: Overview value functions.

be a small number in which cases the walker explores the grid by using the uniform policy π_u. The uniform policy is equal to 0 if a wall is hit and $\frac{1}{4}$ otherwise. Most of the time, the walker uses the deterministic policy as defined in (2.10). This results in **the mixed policy**:

$$\pi_m(s) := \mu \cdot \pi_d + (1 - \mu) \cdot \pi_u \tag{2.12}$$

In the value iteration algorithm in box (5.5) we have $\mu = 0.9$. Continuing, **the mixed reward function** is built in the same way but with a exploration rate $\nu = 0.5$:

$$R_m := (1 - \nu) \cdot R_L + \nu \cdot R_s \tag{2.13}$$

Using the mixed policy (uncomment line 42, and comment line 41) and mixed reward (uncomment line 18,19,20 or 21 depending on the Laplacian used) in the mixed value iteration algorithm found in box (5.5) leads to V_m, **the mixed Laplacian value function**.

2.3 Results of the walking game

In table (2.2), the results are shown when iterating the value iteration algorithm until a minimum path is found. Each direction has the same probability (i.e., $p_N = p_S = p_E = p_W = \frac{1}{4}$). Each algorithm finds a minimum path for the different (adaptive or not) policies and reward functions. When the Laplacians L, L_{rw} and L_{wu} provide the reward function, a convergence in $\sim 10^3$ steps is reached. On the other hand, the sparse reward function and L_{sym} do significantly worse by converging in $\sim 10^4$ steps and a path length of 64. The other three Laplacians find a path of length 40, which equals the length of the minimum path. The paths, as well as the representation of the Fiedler vectors for different Laplacians, can be found in figures (2.21) until (2.15). The most significant difference between adaptive and not adaptive is seen when using the sparse reward function. In V_s, the uniform policy is used and hence can improve a lot when using adaptive learning. Overall adaptive learning is better and will only be used in the next paragraph.

Value function	Policy	adaptive	Reward	Laplacian	Path length	Mean Steps
V_s	π_u	×	R_s	-	5158	$4.98 \cdot 10^4$
V_s	π_u	✓	R_s	-	11208	$4.12 \cdot 10^4$
V_m	π_m	×	R_m	L	40	$1.19 \cdot 10^3$
V_m	π_m	✓	R_m	L	40	$1.205 \cdot 10^3$
V_m	π_m	×	R_m	L_{sym}	90	$5.48 \cdot 10^4$
V_m	π_m	✓	R_m	L_{sym}	64	$5.57 \cdot 10^4$
V_m	π_m	×	R_m	L_{rw}	40	$4.08 \cdot 10^3$
V_m	π_m	✓	R_m	L_{rw}	40	$4.31 \cdot 10^3$
V_m	π_m	×	R_m	L_{wu}	40	$1.55 \cdot 10^3$
V_m	π_m	✓	R_m	L_{wu}	40	$1.34 \cdot 10^3$

Table 2.2: Different value functions with their path lengths and mean number of steps.

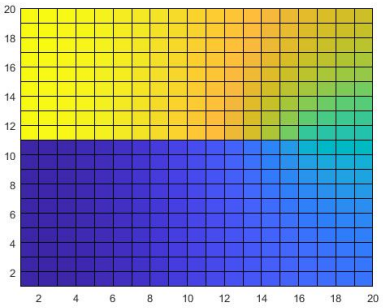


Figure 2.15: Fiedler L.

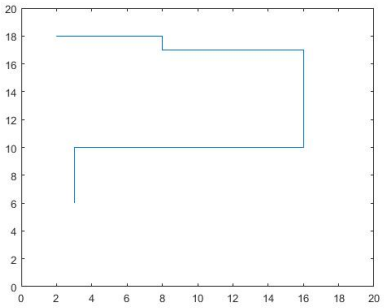


Figure 2.16: Path L of length 40.

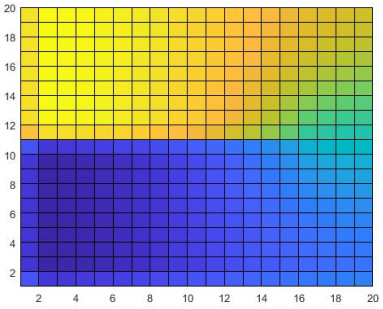


Figure 2.17: Fiedler L_{sym}.

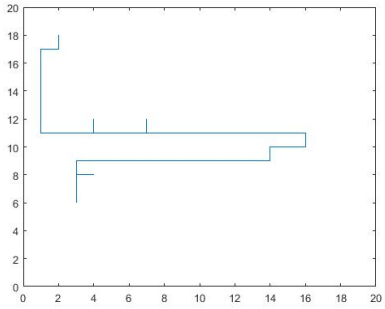


Figure 2.18: Path L_{sym} of length 64.

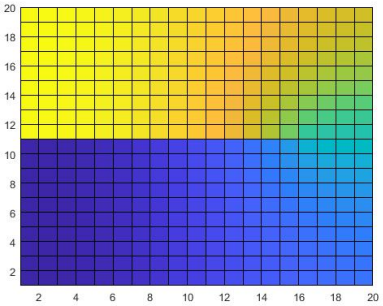


Figure 2.19: Fiedler L_{rw}.

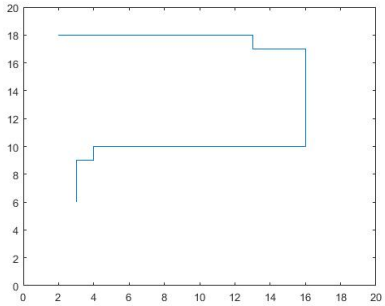


Figure 2.20: Path L_{rw} of length 40.

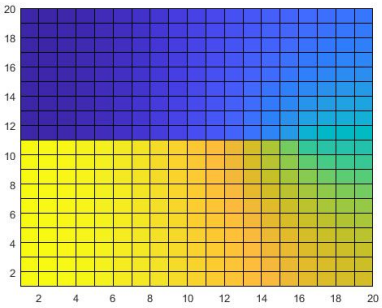


Figure 2.21: Fiedler L_{wu}.

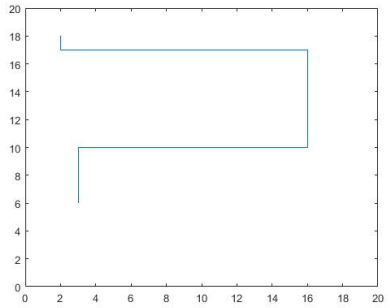


Figure 2.22: Path L_{wu} of length 40.

When using a biased random walker, the differences between the Laplacians are more significant. Different probabilities are used and can be found in the first column of table (2.3). The reward function based on L_{wu} is with no doubt the best. For each direction probability, the value iteration algorithm finds a minimum path of length 40 and does this in the smallest number of steps (all $\sim 10^3$). In terms of terminating, L_{sym} and L_{rw} also do the job. L_{rw} has a termination rate between $\sim 10^3$ and 10^5 steps where L_{sym}'s termination rate is between $\sim 10^4$ and 10^5 steps. Overall, L_{rw} finds a minimum path between 40 and 52 where L_{sym} does not achieve a single time the minimum path length of 40. The smallest path L_{sym} finds is of length 42 and the largest 1426. In figure (2.17) the Fiedler vector of L_{sym} is shown. The areas around the walls of Fiedler L_{sym} are less smooth than the other three Fiedler vectors (see figures (2.15), (2.19) and (2.21)). The basic Laplacian L does not terminate in most cases. In two cases where V_m with L does terminate, it finds a minimum path of length 40 in $\sim 10^3$ steps. The third case terminates in $\sim 10^6$ steps and has a path length of 297029. Looking at the path found in this case, the walkers get sent into a loop several times until it learns that it should not take the loop (see figure (2.23)). $D - P$ calculates the basic Laplacian L and hence is independent of the direction probability. From the results can be concluded that L performs very poorly (first case) or does not terminate at all when the probabilities differ more and more from $\frac{1}{4}$.

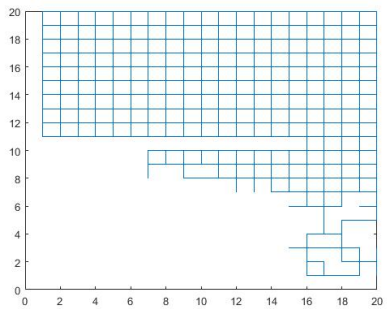


Figure 2.23: Path L of length 297029.

Direction	L	L_{sym}	L_{rw}	L_{wu}
North = 0.27; South = 0.23;				
East = 0.28; West = 0.22;	$6.12 \cdot 10^6$	$1.68 \cdot 10^5$	$1.39 \cdot 10^3$	$1.52 \cdot 10^3$
Path length	297029	42	40	40
North = 0.23; South = 0.27;				
East = 0.22; West = 0.28; -		$2.88 \cdot 10^5$	no convergence	$1.05 \cdot 10^3$
Path length	-	1156	-	40
North = 0.24; South = 0.26;				
East = 0.24; West = 0.26;	-	$5.03 \cdot 10^4$	$2.16 \cdot 10^5$	$1.29 \cdot 10^3$
Path length	-	66	40	40
North = 0.24; South = 0.26;				
East = 0.25; West = 0.25;	$6.95 \cdot 10^3$	$5.02 \cdot 10^4$	$3.08 \cdot 10^4$	$1.27 \cdot 10^3$
Path length	40	82	44	40
North = 0.26; South = 0.24;				
East = 0.25; West = 0.25;	-	$3.70 \cdot 10^5$	$5.25 \cdot 10^3$	$1.20 \cdot 10^3$
Path length	-	44	40	40
North = 0.25; South = 0.25;				
East = 0.26; West = 0.24;	$1.16 \cdot 10^3$	$4.73 \cdot 10^4$	$4.07 \cdot 10^3$	$1.19 \cdot 10^3$
Path length	40	70	40	40
North = 0.25; South = 0.25;				
East = 0.24; West = 0.26;	-	$2.89 \cdot 10^5$	$4.88 \cdot 10^4$	$1.21 \cdot 10^3$
Path length	-	1426	52	40

Table 2.3: Results of different Laplacians with different direction probabilities.

2.4 Approximating Fiedler vector

The results described above show that the Fiedler vector, the second eigenvector of the Laplacian, gives promising results in the walking game. However, the walking game has a finite, even very small, state-space, and the complete graph of the game is known. To be more precise, the exact value function for the walking can be calculated with the Bellman equation in little time. Hence the optimal path can be precisely calculated by using V_{exact} in a value iteration algorithm or by Dijkstra's shortest path algorithm (Dijkstra, 1959). Unfortunately, whenever the state space grows (and thereby the graph), it becomes computationally very expensive to calculate the exact eigenvectors of the Laplacian, or even impossible if the graph is only partly known. Fortunately, a few methods to approximate the Laplacian's eigenvectors are discovered and briefly described next.

Mahadevan (2005) uses the eigenvectors of the actual Laplacian as basis functions (*proto-value functions*) and demonstrates that this speeds up learning with policy iteration. The proto-value functions are task-independent basis functions that form the building blocks for value functions with a given state space. According to Mahadevan (2005), the proto-value functions form geodesically smooth orthonormal basis functions for approximating any value function. However, the eigendecomposition of the actual Laplacian can quickly become quite expensive, and Mahedavan (2005) only shows applicability to discrete state spaces.

Machado et al. (2017) use a diminished matrix for the Laplacian to calculate the eigenvectors. They use (and show) the equivalence of the proto-value functions from Mahadevan (2005) and the spectral decomposition of the successor representation from Stachenfield et al. (2014) to estimate the eigenvectors. They show that the eigenvectors are helpful to construct exploratory behavior. However, even though they use a diminished

matrix and a clever equivalence, the eigendecomposition can also become computationally expensive.

Both papers are represented in the tabular setting, which prevents them from applying to stochastic settings. As a result, it is unclear how good, or even if, these methods perform in a general reinforcement learning setting.

Wu et al. (2018) suggest a computationally efficient way to estimate the eigenvectors of the Laplacian in their paper "*The Laplacian in RL: Learning Representations with Efficient Approximations*." Established on the spectral graph drawing objective, they approximate the eigenvectors. The optimum of the objective leads to the eigenvector representations, contrary to Mahadevan (2005) and Machado et al. (2017), the objective is presented in the general reinforcement learning setting and shows how this can be used in the stochastic setting. So even when the state space is uncountable and is only approachable through sampling, Wu et al. l.'s suggested method is applicable.

For a finite state space, S the Laplacian, according to Wu et al. (2018), can be seen in definition (2.17). This definition is sufficient for the walking game experiment in this thesis. For an infinite state space S, Wu et al. (2018) extend definition (2.17) in the following way:

Definition 2.18. (Wu et al., 2018) If the state set $\mathcal{S}$ is infinite, ρ is the stationary distribution of the transition distribution P^π such that for any measurable $U \subset S$ we have $\rho(U) = \int_S P^\pi(U|v)d\rho(v)$ all with respect to a fixed policy π then the pairwise affinity between two vertices u and v on the graph is

$$L_{wu_2} := \frac{1}{2}\frac{dP^\pi(s|u)}{d\rho(s)}\bigg|_{s=v} + \frac{1}{2}\frac{dP^\pi(s|v)}{d\rho(s)}\bigg|_{s=u}.$$

Now that this definition is available, we can look at the approximation of the eigenvectors. Wu et al (2018) aims to learn the eigen-decomposition embedding ψ, which is a d-dimensional embedding $\psi(u) = [f_1(u), \ldots, f_d(u)]$ and can be learned using a neural network function approximation. The functions f are the eigenfunctions associated with the smallest d eigenvalues of L_{wu_2}.

The states and pairs of states are accessible via sampling even when a model-free reinforcement learning problem is studied. So states u can be sampled from $\rho(u)$ and pairs of u, v can be sampled from $\rho(u)P^\pi(v|u)$. With the choices of the stationary distribution ρ and the Laplacian L_{wu_2} the graph drawing objective can be expressed as an expectation that resolves possible issues if the state space S is huge and/or the graph is not completely known.

Hence, this section explains that there are several techniques available for approximating the eigenvectors of the Laplacian. These approximated eigenvectors work in a more general reinforcement setting, namely if the state space is uncountable and/or the graph is only partly known. The exact calculation of the eigenvectors as described through this chapter and the results leading from them are applicable in countable tabular reinforcement learning problems and gave Mahedavan (2005), Machado et al. (2017), and Wu et al. (2018) the idea of further investigating the eigenvectors of the Laplacian.

3 Conclusion

For tic-tac-toe, a value iteration algorithm is programmed and can be found in box (1.4). This algorithm is tested by using two different opponents. First, the learning player plays against an imperfect player, which prioritizes moves in the following order: winning, blocking, and finally a random move. After a few iterations, the learning player improves his policy to reach an optimum. After 1000 rounds of value iteration, the learning player wins 67,8 % of the time, loses 1,5%, and ties 30,7% of the time against the imperfect player. This optimum does not have a 100% win rate since the imperfect player often plays a random move. After another 1000 rounds, the learning player does not significantly improve. Hence the second opponent was introduced, namely the learning player itself. The learning player improves its policy (*policy 1*) by playing against itself and the previous policy (resulting in *policy A*). Policy A against the imperfect player now has a win rate of 46,6%, ties 53,4%, and never loses! If policy A is tested against policy 1, it also never loses, whereas policy 1 against policy 1 loses 48% of the time. Hence policy A is an improvement in comparison with policy 1. Finally, policy A is tested against itself, and this ends only in draws; no wins or losses were recorded.

For the walking game, different Laplacians are defined, and the Fiedler vectors are used as surrogate reward functions. First, the exact value function is calculated, which is possible due to the small state space. Then, the exact value function is used to compare the other outcomes. The value iteration algorithm for the walking game can be found in box (5.5). First, the sparse reward function is used to make a sparse value function. The sparse value function converges to the exact value function in $\sim 10^7$ steps, and hence we can conclude that the value iteration algorithm we programmed does a good job.

When using the uniform policy (i.e., each direction has the same probability), all four Laplacians: L, L_{sym}, L_{rw} and L_{wu} find a path from start to finish. The minimum path length is 40 and non-unique (established by the exact value function). When the surrogate reward function of the Laplacians L, L_{rw} and L_{wu} is used, a minimum path of length 40 is found in $\sim 10^3$ steps. L_{sym} does significantly worse by converging in $\sim 10^4$ steps and a path length of 64.

When using a biased random walker, the differences between the Laplacians are more significant. The surrogate reward function based on L_{wu} performs the best. Each biased random walker tested converges, finds a minimum path, and does this in the smallest number of steps (all $\sim 10^3$). L_{rw} has a termination rate between $\sim 10^3$ and 10^5 steps and finds a path of lengths between 40 en 52. L_{sym}'s termination rate is between $\sim 10^4$ and 10^5 steps and does not find a path of minimum length (between length 42 and 1426). The basic Laplacian L does not terminate in most cases. In the two (out of seven) cases where it does terminate, it finds a minimum path of length 40 in $\sim 10^3$ steps. From the results can be concluded that L performs very poorly or does not terminate at all when the probabilities differ more and more from $\frac{1}{4}$.

4 Appendix: Tic-tac-toe

```
function B = state2board(state);

B = zeros(length(state),9);

for k=9:-1:1,
    B(:,k) = mod(state,3);
    acstate = (state-B(:,k))/3;
end

return;
```

Box 4.1: State to board function

```
function state = board2state(B);

state = zeros(size(B,1),1);

for k=1:9,
    state = state*3 + B(:,k);
end

return;
```

Box 4.2: Board to state function

```
function player = turn(state);

B = state2board(state);

n1 = sum(B==1,2);
n2 = sum(B==2,2);

player = 2*(n1==(n2+1)) + (n1==n2);

end
```

Box 4.3: Turn function

```
function check = reachableState(state);

player = turn(state); % player == 0 if this state is unreachable by ...
    taking turns starting with player 1
wins1 = checkWin(state,1);
wins2 = checkWin(state,2);

% state is 'reachable' if it can be reached by alternating play
% starting with player 1 and if at most 1 player has won

% note that in alternating play, player 1 may occupy at most 5 cells, ...
    player 2 at most 4 cells,
% so possible full-board states are: draw, single-win player 1,
% single win player 2, double win player 1

% a state in which a player has won and again receives a turn, is also
% unreachable

if player==0 | (wins1>0 & wins2>0)
    check = false;

elseif (wins1>0 & player==1) | (wins2>0 & player==2)
    check = false;

else
    check = true;

end

end
```

Box 4.4: Reachable State function

```
function Nwins = checkWin(state,check_player);

board = state2board(state);

ind = [1 2 3; 4 5 6; 7 8 9; 1 4 7; 2 5 8; 3 6 9; 1 5 9; 3 5 7];

Nwins = length(find(sum(board(ind)==check_player,2)==3));

end
```

Box 4.5: Check Win function

```
function isnomove = noMove(state)

isnomove = isempty(find(state2board(state)==0));

end
```

Box 4.6: No Move function

```
function [newstate,reward] = opponent(state,action,OPPfun,Pi,singleOutput);

player = turn(state);

between_state = board2state(state2board(state) + action);
if between_state == 0
    between_state = 3^9;
end

if checkWin(between_state,player)
    newstate = -1;
    reward = 1;
elseif noMove(between_state) % no plays left: draw
    newstate = -1;
    reward = 0;
else
    newstate = OPPfun(between_state,Pi);
    reward = zeros(size(newstate));

    for j=1:length(newstate)
        if checkWin(newstate(j),3-player)
            reward(j) = -1;
        elseif noMove(newstate(j)) % no plays left
            reward(j) = 0;
        else
            reward(j) = 0;
        end
    end

    if singleOutput
        j = randi(length(newstate));
        newstate = newstate(j);
        reward = reward(j);
    end
end
end
```

Box 4.7: Opponent function

```
function newstate = imperfect_player(state,Pi);

[p1,p2,i1,i2] = critState(state);

B = state2board(state);

player = turn(state);

a = zeros(1,9);

if player==2,

    if ¬isempty(i2) % chance to win
        a(i2) = 2;
    elseif ¬isempty(i1) % chance to block
        a(i1) = 2;
    else % random play
        ind = find(B==0);
        a(ind) = 2;
    end

else

    if ¬isempty(i1) % chance to win
        a(i1) = 1;
    elseif ¬isempty(i2) % chance to block
        a(i2) = 1;
    else % random play
        ind = find(B==0);
        a(ind) = 1;
    end

end

ind = find(a);
Bold = B;
newstate = zeros(length(ind),1);

for j=1:length(ind);

    B = Bold;
    B(ind(j)) = a(ind(j));
    newstate(j) = board2state(B);

end

end
```

Box 4.8: Imperfect player function

```
function [p1,p2,i1,i2] = critState(state);

B = state2board(state);

ind = [1 2 3; 4 5 6; 7 8 9; 1 4 7; 2 5 8; 3 6 9; 1 5 9; 3 5 7];

BB = B;
BB(B==2)=4;

rcdsum = sum(BB(ind),2);

p1 = find(rcdsum==2);
p2 = find(rcdsum==8);

el1 = ind(p1,:);
el2 = ind(p2,:);

a1 = find(B(el1)==0);
a2 = find(B(el2)==0);

i1 = el1(a1);
i2 = el2(a2);
end
```

Box 4.9: Critical state function

```
function newstate = playPolicy(state,Pi);

player = turn(state);

B = state2board(state);

ind_pi = find(Pi(state,:))';
newstate = zeros(size(ind_pi));

for j=1:length(ind_pi),
    a = zeros(1,9);
    a(ind_pi(j)) = player;
    newstate(j) = board2state(B + a);
end

end
```

Box 4.10: Opponent policy function

```
function [win,lose,draw] = testPolicy (Pi1,Pi2,OPPfun,numGames);

win = 0;
lose = 0;
draw = 0;

for n=1:numGames,

    state = 3^9;
    first_player = (rand < 0.5) + 1;

    if first_player==2

        saveState = zeros(1,5);
        action = zeros(1,9);
        [state,reward] = opponent(state,action,OPPfun,Pi2,true);
        saveState(1) = state;

        for moves = 1:4,
            ind = find(Pi1(state,:));
            action = zeros(1,9);
            action(ind(randi(length(ind)))) = 2;
            [state,reward] = opponent(state,action,OPPfun,Pi2,true);
            saveState(moves+1) = state;

            if reward≠0 | state==-1
                break;
            end
        end

    else
        for moves = 1:5,
            ind = find(Pi1(state,:));
            action = zeros(1,9);
            action(ind(randi(length(ind)))) = 1;
            [state,reward] = opponent(state,action,OPPfun,Pi2,true);

            if reward≠0 | state==-1
                break;
            end

        end
    end

    if reward == 1
        win = win + 1;
    elseif reward == -1
        lose = lose + 1;
    elseif reward == 0
        draw = draw + 1;
    else
        disp(reward);
        disp('Hmmm... should not be here');
    end
end
end
```

Box 4.11: Test Policy function

5 Appendix: Walking game

```
function Pi = randWalk(A,M,K);

    D = diag(sum(A,2));
    Pi = D\A;

end
%
```

Box 5.1: Random walk policy

```
function S = pathDet(Pi,start,finish)
    s = start;
    S = [s];
    while s ≠ finish,
        [mx,imax] = max(Pi(s,:));
        s = imax;
        if find(S==s)
            break;
        end
        S = [S s];
    end
end
```

Box 5.2: Path determination

```
function [P,D,L,X,Lambda] = topo(M,K);

% MxK earest neighbor connection graph:

North = 0.25; South = 0.25; East = 0.25; West = 0.25;

Acol = South * diag(ones(M-1,1),1) + North * diag(ones(M-1,1),-1);
Arow = East * diag(ones(K-1,1),1) + West * diag(ones(K-1,1),-1);

A = kron(eye(K),Acol) + kron(Arow,eye(M));

% insert walls by breaking connections
for j=1:round(0.75*K);
    i1 = M/2;
    i2 = i1+1;

    k1 = (j-1)*M + i1;
    k2 = (j-1)*M + i2;

    A(k1,k2) = 0;
    A(k2,k1) = 0;
end

%% Weight
D = diag(sum(A,2));
L = D - A;
[X,Lambda] = eig(L);
Lambda = diag(Lambda);
X2 = X(:,2);

%%  Sym
Y = diag(D);
L_sym = diag(Y.^(-1/2)) * A * diag(Y.^(-1/2));
[X_sym,Lambda_sym] = eig(L_sym); Lambda_sym = diag(Lambda_sym);

[ll_sym,ind_sym] = sort((-Lambda_sym));
X2_sym = X_sym(:,ind_sym(2));

%% Rw
L_rw = diag(Y.^(-1/2)) * L_sym * diag(Y.^(1/2));
[X_rw,Lambda_rw] = eig(L_rw); Lambda_rw = diag(Lambda_rw);

[ll_rw,ind_rw] = sort((-Lambda_rw));
X2_rw = X_rw(:,ind_rw(2));

%% Wu
I = eye(M*K);
B = inv(D) * A;
rho = null(B-I);
D = A/diag(rho');
D = (D + D')/2;
L = diag(sum(D,2)) - D;

[X_wu,Lambda_wu] = eig(L);
Lambda_wu = diag(Lambda_wu); [ll_wu,ind_wu] = sort((-Lambda_wu));
X2_wu = X_wu(:,ind_wu(2));

%% Transition probability matrix
P = A;

end
```

Box 5.3: Topography of grid and Laplacians

```
function V = Vexact (Pi,A,reward,finish);
%
%   function V = Vexact (Pi,A,finish);
%
%   Computes the value function V, given:
%   A = adjacency matrix (NxN matrix, for N gridpoints)
%   Pi = Policy matrix (same sparsity structure as A)
%   reward = Reward vector (N-vector)
%   finish = objective gridpoint
%

    N = size(A,1);  % number of states

    V = zeros(N,1);
    Vold = V;

    gamma = 0.9;
    maxIter = 110;
    tol = 1e-10;

    for iter = 1:maxIter,

        Visited = false(N,1);
        ind = finish;
        Visited(finish) = true;

        while find(¬Visited)
            allS = [];
            for sprime = ind;
                S = find(A(:,sprime)');
                for i = S
                    if ¬Visited(i)
                        V(i) = Pi(i,:)*(reward + gamma*V);
                        Visited(i) = true;
                    end
                end
                allS = [allS S];
            end
            pcolor(flipud(reshape(log(V+1e-10),20,20))); shading flat; ...
                drawnow;
            ind = unique(allS);
        end

        resid = norm(Vold-V,inf);
        if mod(iter,10)==0
            disp(iter);
            disp(resid);
        if resid < tol,
            disp(resid);
            break;
        end
        Vold = V;

    end

end
```

Box 5.4: V_{exact} walking game

```
function [V,Nsamp,steps,path] = ...
    Viter(V,Nsamp,Pi,A,reward,Nepisodes,finish,start);

    N = size(A,1);
    gamma = 0.9;
    steps = 0;

    for episode = 1:Nepisodes

        state = start;
        path = state;

        while state≠finish

            steps = steps + 1;

            spind = find(A(state,:));
            p = cumsum(Pi(state,spind));
            action = find(rand()<p,1);
            sprime = spind(action);

            if Nsamp(sprime)>0
                Vprime = V(sprime)/Nsamp(sprime);
            else
                Vprime = 0;
            end
            V(state) = V(state) + (reward(sprime) + gamma*Vprime);
            Nsamp(state) = Nsamp(state) + 1;

            state = sprime;

            path = [path state];

        end

        Nsamp(finish) = Nsamp(finish)+1;

    end

end
```

Box 5.5: Value iteration algorithm walking game

```
M = 20;
K = 20;

[A,D,L,X,Lambda] = topo(M,K);
Pi = randWalk(A,M,K);                    % Uniform random(walk) policy

finish = 55;                             % [3,6]
start = 23;                              % [2,18]
reward = zeros(M*K,1);
reward(finish) = 1;                      % sparse reward

Ve = Vexact(Pi,A,reward,finish);         % iterated Value function

Nepi = 4000;                             % Number of episodes
steps = zeros(Nepi,1);                   % Number of steps per episode

Vsum = zeros(M*K,1);                     % running sum Value function
Nsamp = zeros(M*K,1);                    % number of visits to each ...
    state, V=Vsum./Nsamp
err = zeros(Nepi,1);

for ep = 1:Nepi,

    [Vsum,Nsamp,steps(ep)] = Viter(Vsum,Nsamp,Pi,A,reward,1,finish,start);
    V = Vsum./Nsamp;
    err(ep) = norm(V - Ve);

end

loglog(cumsum(steps),err);
set(gca,'fontsize',14);
xlabel 'Number of steps, random walker'
ylabel '|| V - V_e ||'
```

Box 5.6: Calculating convergence sparse value function to exact sparse value function.

```
%% Domain parameters

% Grid dimensions
M = 20;
K = 20;

% Connection graph: Adjacency matrix, start and finish (objective)
[A,D,L,X,Lambda] = topo(M,K);
finish = 55;
start = 23;

%% Reward functions:
% Sparse reward function
Rsparse = zeros(M*K,1);
Rsparse(finish) = 1;

% Eigenvector reward function L, uncomment for Laplacian needed
R2 = sqrt((X(:,2)-X(finish,2)).^2);              %L
% R2 = sqrt((X2_sym-X2_sym(finish)).^2);         %L_sym
% R2 = sqrt((X2_rw-X2_rw(finish)).^2);           %L_rw
% R2 = sqrt((X2_wu-X2_wu(finish)).^2);           %L_wu
R2 = R2/max(R2);
R2 = 1 - R2;

% Mixed reward function
nu = 0.5;
Rmix = nu*R2 + (1-nu)*Rsparse;

%% Policies
%  Uniform random(walk) policy
Pi_u = randWalk(A,M,K);

% Reward-maximizing deterministic path:
Pi_r = detPol(Rmix,A,finish);

% Mixed policy function
mu = 0.9;
Pi_mix = mu*Pi_r + (1-mu)*Pi_u;

% Choice of policy (Pi_u or Pi_mix)
%Pi = Pi_u;
Pi = Pi_mix;

%% Exact Value function, based on iterated backup strategy (Bellman's ...
    equation)
% Ve = Vexact(Pi_u,A,Rsparse,finish);  % iterated Value function
% Se = pathDet(detPol(Ve,A,finish),start,finish);
% Ve_path = Ve(Se);

%% Learn value function
Nepi = 4000;    % Number of episodes
TotalEpisodes = 0;
TotalSteps = 0;
Nsamples = 100;

Continues next page
```

Box 5.7: Main part 1

```
% Repeat several times
for sample = 1:Nsamples,

    Vsum = zeros(M*K,1);  % running sum Value function
    Nsamp = zeros(M*K,1);  % number of visits to each state, V=Vsum./Nsamp
    err = zeros(Nepi,1);
    tol = 1e-8;
    steps = zeros(Nepi,1);  % Store number of steps per episode

    for ep = 1:Nepi,

        % Compute one value iteration based on an episode (start to finish)
        % using the mixed Policy
        [Vsum,Nsamp,steps(ep),path] = ...
            Viter(Vsum,Nsamp,Pi,A,Rsparse,1,finish,start);
        V = Vsum./Nsamp;

        % Note the value function V will not converge to Vex (even ...
            along the
        % optimal path and using Rsparse) because the values are
        % policy-dependent for stochastic policies

        % What is a resonable error criterion?  In fact, the way the ...
            problem is
        % posed (i.e. sparse reward), any minimal-length path is optimal.
        % Consequently, any Value function is acceptable, as long as it ...
            yields
        % a determinstic path of shortest length.

        Pi_V = detPol(V,A,finish);
        S = pathDet(Pi_V,start,finish);


        % Uncomment (next 3 lines) for adaptive policy (starting with ...
            uniform random)
        if min(V)>0
            Pi = (1-mu)*Pi_u + mu*Pi_V;
        end

        if S(end) == finish
            err(ep) = length(S) - length(Se);
        else
            err(ep) = length(path)-length(Se);
        end

        if err(ep)<tol
            break;
        end

    end
    if ep≥Nepi,
        disp('no convergence to minimum path');
    end

    TotalEpisodes = TotalEpisodes + ep;
    TotalSteps = TotalSteps + sum(steps);

end

MeanEpisodes = TotalEpisodes/Nsamples
MeanSteps = TotalSteps/Nsamples
```

Box 5.8: Main part 2

Bibliography

[Sha50] Claude E Shannon. "XXII. Programming a computer for playing chess". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275.

[Mar54] A. A. Markov. "Theory of Algorithms." In: (1954).

[Bel57] Richard Bellman. "Dynamic programming". In: *Press Princeton, New Jersey* (1957).

[Dij+59] Edsger W Dijkstra et al. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[How60] Ronald A Howard. "Dynamic programming and markov processes." In: (1960).

[Min61] Marvin Minsky. "Steps toward artificial intelligence". In: *Proceedings of the IRE* 49.1 (1961), pp. 8–30.

[Whi63] Douglas J White. "Dynamic programming, Markov chains, and the method of successive approximations". In: *Journal of Mathematical Analysis and Applications* 6.3 (1963), pp. 373–376.

[Sch67] Donald G Schultz. "State functions and linear control systems". In: *McGraw-Hill Book Company* (1967).

[Fie89] Miroslav Fiedler. "Laplacian of graphs and algebraic connectivity". In: *Banach Center Publications* 25.1 (1989), pp. 57–70.

[Wat89] Christopher John Cornish Hellaby Watkins. "Learning from delayed rewards". In: (1989).

[All+94] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

[Bar95] Andrew G Barto. "Reinforcement learning and dynamic programming". In: *Analysis, Design and Evaluation of Man–Machine Systems 1995*. Elsevier, 1995, pp. 407–412.

[BT96] Dimitri P Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[CG97] Fan RK Chung and Fan Chung Graham. *Spectral graph theory*. 92. American Mathematical Soc., 1997.

[Dre02] Stuart Dreyfus. "Richard Bellman on the birth of dynamic programming". In: *Operations Research* 50.1 (2002), pp. 48–51.

[NJW02] Andrew Y Ng, Michael I Jordan, and Yair Weiss. "On spectral clustering: Analysis and an algorithm". In: *Advances in neural information processing systems*. 2002, pp. 849–856.

[Kor03] Yehuda Koren. "On Spectral Graph Drawing". In: *Proceedings of the 9th Annual International Conference on Computing and Combinatorics*. COCOON'03. Big Sky, MT, USA: Springer-Verlag, 2003, pp. 496–508. ISBN: 3540405348.

[Mah05] Sridhar Mahadevan. "Proto-value functions: Developmental reinforcement learning". In: *Proceedings of the 22nd international conference on Machine learning*. 2005, pp. 553–560.

[VW12] Martijn Van Otterlo and Marco Wiering. "Reinforcement learning and markov decision processes". In: *Reinforcement learning*. Springer, 2012, pp. 3–42.

[Gal13] Jean Gallier. "Notes on elementary spectral graph theory. applications to graph clustering using normalized cuts". In: *arXiv preprint arXiv:1311.2492* (2013).

[SBG14] Kimberly L. Stachenfeld, Matthew M. Botvinick, and Samuel J. Gershman. "Design Principles of the Hippocampal Cognitive Map". In: NIPS'14 (2014), pp. 2528–2536.

[SB14] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT Press, 2014. ISBN: 9780262039246.

[Jas17] Arnout Jaspers. "Van nul tot bovenmenselijk: in drie dagen Go spelen". In: *Kennislink* (2017).

[Nac+18] Ofir Nachum et al. "Data-Efficient Hierarchical Reinforcement Learning". In: *CoRR* abs/1805.08296 (2018). arXiv: 1805.08296. URL: http://arxiv.org/abs/1805.08296.

[WTN18] Yifan Wu, George Tucker, and Ofir Nachum. "The Laplacian in RL: Learning Representations with Efficient Approximations". In: *CoRR* abs/1810.04586 (2018). arXiv: 1810.04586. URL: http://arxiv.org/abs/1810.04586.

[Dur19] Rick Durrett. *Probability: theory and examples*. Vol. 49. Cambridge university press, 2019.

[GQ20] Jean Gallier and Jocelyn Quaintance. *Linear Algebra and Optimization with Applications to Machine Learning: Volume I: Linear Algebra for Computer Vision, Robotics, and Machine Learning*. World Scientific, 2020.

[Bha] Shweta Bhatt. *5 Things You Need to Know about Reinforcement Learning*. URL: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html. (accessed: 06.2021).

[Dou] Brian Douglas. *What is Reinforcement Learning?* URL: https://nl.mathworks.com/videos/reinforcement-learning-part-1-what-is-reinforcement-learning-1551974943006.html. (accessed: 07.2021).

[Jin] Hong Jing. *Reinforcement Learning - The Value Function*. URL: https://towardsdatascience.com/reinforcement-learning-value-function-57b04e911152. (accessed: 07.2021).

[Nai] Surag Nair. *A Simple Alpha(Go) Zero Tutorial*. URL: https://web.stanford.edu/~surag/posts/alphazero.html. (accessed: 06.2021).

[Pap] Papergames.io. *Play tic-tac-toe*. URL: https://papergames.io/en/tic-tac-toe. (accessed: 07.2021).

[Tor] Jordi Torres. *The Bellman Equation*. URL: https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7. (accessed: 07.2021).