Emergency Trajectory Management

for Transport Aircraft

by

Victor Snoeij

to obtain the degree of Master of Science
in Aerospace Engineering

at the Delft University of Technology,

to be defended publicly on Thursday July 28, 2016 at 14:00.

Supervisor :	Dr. ir. H. G. Visser	
Thesis committee:	Dr. ir. H. G. Visser,	TU Delft
	Ir. P. C. Roling,	TU Delft
	Dr. ir. E. Mooij	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

Over the years gradually flight management systems have been added to airplanes which reduce pilot workload and increase safety. These systems however do not provide an adequate response when in an emergency situation the aircraft looses all thrust. A system that creates glide trajectories to airports that are reachable would enable pilots to focus their attention on stabilizing and regaining control in an in-flight emergency situation.

The objective of this study is to develop such an emergency flight system, that can generate glide trajectories from the position at which the emergency starts to all reachable runways. By first determining the maximum range of the aircraft, a footprint is defined in which the reachable runways are identified. To reduce risk on board as well as on the ground, the trajectories generated must be balanced between flying over populated areas effectively increasing the risk of loss of life on ground or avoiding populated areas possibly increasing the risk of loosing control of the aircraft by having to be airborne for a longer period of time. The trajectories to all reachable runways are ranked based on several airport and runway quality factors and are presented to the pilot to choose from.

The trajectories are generated by an Approximate Dynamic Programming (ADP) algorithm which is commonly used in robotics for path planning. In this research it is investigated if it can also be applied in the field of aeronautics for the purpose of an on line emergency trajectory planner. In a test scenarios, trajectories were generated by several ADP variants and compared with a benchmark trajectory created by the A* algorithm. This algorithm generates optimal obstacle avoiding trajectories providing certain conditions are met. From the test it resulted that Q-learning performed the best and was therefore chosen to generate the trajectories for the emergency trajectory planner. When the trajectories are generated by the ADP algorithm, they are used as a reference path to create a smoothed trajectory in the horizontal as well as in the vertical plane.

With the trajectory generating algorithm determined, the emergency trajectory planner is used in a scenario in which the aircraft has to generate trajectories to the reachable airports in the footprint. The results show that Q-learning generates shorter trajectories than the benchmark to airports which are surrounded by obstacles. The trajectories for this scenario vary between 55% and 74% compared to the A*. The Q-learning trajectories are shorter because it balances between a longer route resulting in a higher summed distance cost or a shorter route but receiving penalties. The Q-learning trajectories to airports which are not surrounded by obstacles vary between 94% and 112% compared with the A*. The fact that the Q-learning trajectories are longer than the A* is in agreement with the theory. The shorter routes are due to the fact that where the A* avoids all obstacles, Q-learning is able to go through the obstacles resulting in shorter trajectories.

Although approximate dynamic programming is a valuable algorithm for path planning in robotics, it proves less suitable to create trajectories on line in emergency situations. While it successfully generated trajectories to the reachable runways, the time the algorithm needs to create these trajectories is too long to be able to implement it in emergency situations. By using a stopping criterion to converge the value function, optimality of the results are not guaranteed which is an undesirable property in emergency situations. Also by not taking into account flight dynamics when generating the trajectory, obstacles might not be avoided optimally.

Acknowledgement

Met het schrijven van deze scriptie sluit ik mijn studietijd aan de Technische Universteit Delft af. In deze acknowledgment wil ik mij graag richten tot de mensen die mij geholpen hebben dit eindpunt te bereiken. Ik wil allereest Dries Visser bedanken voor het begeleiden van dit onderzoek. Daarnaast wil ik de faculteit Lucht- en Ruimtevaart en in speciaal de afdeling ATO bedanken voor het faciliteren van deze prachtige master. Ik wil verder ook de (oud) bewoners en reguliere bezoekers van 3.13 bedanken voor de mooie koffie momenten en discussies over elk mogelijk onderwerp.

Graag wil ik ook de leden van WB2007 bedanken voor de gezellige tijd die meteen begon bij aankomst in Delft. Het maakte het allemaal een stuk mooier om in Delft te studeren. Voor het doorlezen van mijn scriptie en mij voorzien van tips, wil ik graag Hector en Floris bedanken.

Ik wil ook mijn familie bedanken voor de jarenlange steun die ik van ze gekregen heb en dat ze altijd voor mij klaar staan. Zonder jullie had ik nooit vrijgeleide a38 gevonden.

Finalmente quiero agradecer mi novia Patricia. Gracias por aguantar todos mis tonterias durante mi pfc.

List of Figures

1.1 Fatal accidents worlwide commercial jet fleet 2005 through 2014 (Boeing, 2014) 2

1.2 Generated footprint (Atkins et al., 2006) 2

1.3 Dubins path (Atkins et al., 2006) 3

1.4 Footprint (Coombes et al., 2013) 4

1.5 The high key low key technique (Coombes et al., 2013) 4

2.1 Architecture of the trajectory planner 8

2.2 Map with footprint 9

2.3 K-means 11

2.4 Obstacle defining 11

2.5 Comparing original and adjusted area 12

2.6 Adapted from Fernandes de Oliveira and Büskens (2013) 13

2.7 Quality measures for runway utility computation (Atkins et al., 2006) 16

3.1 Deterministic Markov Decision Process 18

3.2 Backup diagrams Policy Iteration (Sutton and Barto, 1998) 20

3.3 Backup diagrams Value Iteration (Sutton and Barto, 1998) 21

3.4 Example 1 (Visser, 2015) 22

3.5 Optimal solution example 1 22

3.6 Policy Iteration 24

3.7 Policy Iteration optimal solution example 1 24

3.8 Value Iteration 26

3.9 K-steps return (Sutton and Barto, 1998) 27

3.10 Backup diagram TD 28

3.11 TD greedy policy sample path example 1 29

3.12 TD backup diagrams for example 1 29

3.13 Performance k-step TD methods (Sutton and Barto, 1998) 29

3.14 Backup diagram TD ϵ greedy . . . 30
3.15 Backup diagrams Sarsa & Q-learning (Sutton and Barto, 1998) . . . 31
3.16 Forward view . . . 32
3.17 Accumulating and replacing traces (Singh and Sutton, 1996) . . . 32
3.18 Performance on-line TD(λ) (Sutton and Barto, 1998) . . . 34
3.19 Greedy policy sample path scenario I . . . 36
3.20 Greedy policy sample path scenario II . . . 37
3.21 TD ϵ-greedy . . . 38
3.22 TD ϵ-greedy eligibility traces . . . 39
3.23 Random generated comparison grid . . . 42
3.24 Possible actions . . . 42
3.25 A* path scenario I . . . 43
3.26 Comparing ϵ Sarsa scenario I . . . 44
3.27 Comparing ϵ Q-learning scenario I . . . 45
3.28 Comparing Sarsa and Q-learning scenario I . . . 46
3.29 A* path scenario II . . . 48
3.30 Sarsa and Q-learning scenario II . . . 49
3.31 Sarsa and Q-learning scenario II adjusted scale . . . 50

4.1 Possible actions ADP model . . . 54
4.2 Example 2 segments trajectory . . . 54
4.3 Influence of penalties on trajectories . . . 55

5.1 Map with footprint . . . 58
5.2 Woensdrecht route . . . 58
5.3 Volkel route . . . 59
5.4 Kempen route . . . 59
5.5 Airport routes . . . 60
5.6 Direct routes . . . 61
5.7 Rotterdam route . . . 62
5.8 Schiphol route . . . 62
5.9 Quality measures for runway utility computation (Atkins et al., 2006) . . . 63

A.1 Schiphol route 67
A.2 Rotterdam horizontal route 68
A.3 Rotterdam vertical route 68
A.4 Volkel horizontal route 68
A.5 Volkel vertical route 69
A.6 Kempen horizontal route 69
A.7 Kempen vertical route 69
A.8 Gilze Rijen horizontal route 70
A.9 Gilze Rijen vertical route 70
A.10 Woensdrecht horizontal route 70
A.11 Woensdrecht vertical route 71

List of Algorithms

1 Policy iteration (Sutton and Barto, 1998) 21

2 Value iteration (Sutton and Barto, 1998) 21

3 Tabular TD(0) (Sutton and Barto, 1998) 28

4 Sarsa (Sutton and Barto, 1998) 30

5 Q-Learning (Sutton and Barto, 1998) 31

6 On-line tabular TD(λ) (Sutton and Barto, 1998) 33

7 Tabular Sarsa (λ) (Sutton and Barto, 1998) 34

8 Q-learning (Sutton and Barto, 1998) 53

List of Tables

2.1 Boeing 737 model (clean configuration) 8

2.2 Population density factors 11

3.1 State values Policy Iteration 23

3.2 State values Value Iteration 25

3.3 Greedy policy sample path scenario I 36

3.4 Greedy policy sample path scenario II 37

3.5 Q-learning values ϵ-greedy 38

3.6 Q-learning values ϵ-greedy 39

3.7 Eligibility traces 40

3.8 Value function with eligibility traces 40

3.9 Scenario I simulation time 47

3.10 Scenario II simulation time 48

5.1 Ranked runways 64

A.1 Schiphol Airport runways 71

A.2 Teuge Airport runway 71

A.3 Maastricht Aachen Airport runway 71

A.4 Kempen Airport runway 72

A.5 De Kooy Airfield runway 72

A.6 Eindhoven Airport runway 72

A.7 Enschede Airport Twente runway 72

A.8 Groningen Airport Eelde runways 72

A.9 Lelystad Airport runway 73

A.10 Rotterdam The Hague Airport runway 73

A.11 Deelen Air Base runway 73

A.12 Gilze-Rijen Air Base Runways 73

A.13 Leeuwarden Air Base runways 73
A.14 Volkel Air Base runways 74
A.15 Lieutenant General Best Barracks runway 74
A.16 Woensdrecht Airbase runway 74
A.17 Atmospheric parameters 74

Contents

Abstract iii

Acknowledgement v

List of Figures vii

List of algorithms . . . xi

List of Tables xiii

1 Introduction 1

1.1 Background . . . 1

1.2 Literature . . . 2

1.3 Research Definition . . . 6

1.3.1 Objectives . . . 6

1.3.2 Research Questions . . . 6

1.4 Report Structure . . . 6

2 The Trajectory Planner 7

2.1 Footprint . . . 7

2.2 Trajectory Generation . . . 10

2.2.1 Population Avoidance . . . 10

2.2.2 Horizontal Trajectories . . . 13

2.2.3 Energy Dissipation Sector . . . 13

2.3 Dynamic Model . . . 14

2.4 Trajectory Ranking . . . 15

3 Dynamic Programming 17

3.1 Dynamic Programming . . . 17

3.1.1 Markov Decision Process . . . 17

3.1.2 Value Functions . . . 18

3.1.3 Policy Iteration . . . 19
3.1.4 Value Iteration . . . 20
3.1.5 Example . . . 22
3.2 Approximate Dynamic Programming . . . 26
3.2.1 Temporal Difference Learning . . . 26
3.2.2 Sarsa & Q-learning . . . 30
3.2.3 TD Lambda . . . 31
3.2.4 Sarsa λ & Q λ . . . 33
3.2.5 Learning Rate . . . 35
3.2.6 Example . . . 36
3.3 Selecting the route planning algorithm . . . 40
3.3.1 A* Algorithm . . . 41
3.3.2 Experiment . . . 41
3.3.3 Scenario I . . . 43
3.3.4 Scenario II . . . 48

4 Model Specifications **53**

4.1 Target Trajectory Generation . . . 53
4.2 Penalty Value . . . 55

5 Results **57**

5.1 Horizontal Trajectories . . . 58
5.2 Vertical Trajectories . . . 61
5.3 Ranking . . . 63

6 Conclusions & Recommendations **65**

6.1 Conclusions . . . 65
6.2 Recommendations . . . 66

A Appendix **67**

A.1 Trajectory results . . . 67
A.2 Airports . . . 71
A.2.1 Schiphol (EHAM) . . . 71
A.2.2 Teuge Airport (EHTE) . . . 71

A.2.3 Maastricht Aachen Airport (EHBK) . . . 71
A.2.4 Kempen Airport (EHBD) . . . 72
A.2.5 De Kooy Airfield (EHKD) . . . 72
A.2.6 Eindhoven Airport (EHEH) . . . 72
A.2.7 Enschede Airport Twente (EHTW) . . . 72
A.2.8 Groningen Airport Eelde (EHGG) . . . 72
A.2.9 Lelystad Airport (EHLE) . . . 73
A.2.10 Rotterdam The Hague Airport (EHRD) . . . 73
A.2.11 Deelen Air Base (EHDL) . . . 73
A.2.12 Gilze-Rijen Air Base (EHGR) . . . 73
A.2.13 Leeuwarden Air Base (EHLW) . . . 73
A.2.14 Volkel Air Base (EHVK) . . . 74
A.2.15 Lieutenant General Best Barracks (EHDP) . . . 74
A.2.16 Woensdrecht Airbase (EHWO) . . . 74
A.3 Atmospheric Parameters . . . 74

Bibliography **75**

1 Introduction

1.1. Background

On the 23rd of July 1983 Air Canada flight 143, also known as the 'Gimli Glider', lost all engines mid route due to fuel mismanagement resulting in insufficient fuel on board. The pilots were able to glide the aircraft to a nearby airport landing it safely, thanks to the gliding experience of the captain (FSF, 2015). During the accident investigation several attempts by other crews who were given the same circumstances in a simulator resulted in crashes indicating that the 'Gimli Glider' could have ended fatal.

Another example of landing after full loss of thrust is Air Transat flight 236. While flying over the Atlantic Ocean the aircraft lost all engine power 65 miles (120 km) out of the nearest runway on the Azores. The aircraft managed to reach the island gliding, while the aircraft suffered structural damage there were no fatal casualties (GPIAA, 2001).

According to the 24nd Joseph T. Nall report (AOPA, 2015) in 2012, 146 of the in total 1162 accident (13 %) with non commercial fixed wing aircraft suffered from loss of thrust. For part 135 charter and cargo aircraft 3 of the 26 (11 %) accidents were caused by loss of thrust. The causes for these accidents were pilot related (erroneous fuel management) and mechanical (power plant, fuel systems). Both causes result in loss of thrust that, if not handled well, can create hazardous situations.

For commercial jet airplanes it can be deduced from figure 1.1 that in the period between 2005 & 2015 accidents in the category *System/Component Failure or Malfunction* (SCF-PP) can partly be attributed to loss of thrust. The accidents in this category are caused by system and component failure/malfunctioning which also includes the power plant (engine). In this category in total there were 165 fatalities from which 12 not on board. Loss of thrust might not be the biggest risk in non-commercial flights but pilots should be aware of the occurrence of forced landing due to complete or partial loss of thrust.

If the engines of an aircraft shut down, pilots immediately follow checklists to restart the engines. If they cannot be restarted, the aircraft can be provided with the necessary power for the essential systems (flight critical instrumentation, flight controls and hydraulics), by a ram air turbine (RAT). This is a small turbine that generates power from the air stream due to the speed of the aircraft and is deployed below the hull of the aircraft. The next step to safely land the aircraft would be to find a runway to land.

The goal of this research is to explore the possibility of creating a system that can generate gliding routes on line from the current aircraft position to reachable airports taking into account the degraded performance of the aircraft due to power loss. This system would enable pilots to focus their attention on stabilizing and regaining control in an in-flight emergency situation. To increase the glide range and with that the number of reachable airports, the planner should optimize the glide path. If pilots are unable to restart the engines, the trajectory planner would already have, if possible, multiple routes generated for them to follow, increasing the

possibility of a successful landing.
In figure 1.1 next to onboard fatalities, external fatalities are specified. In order to reduce on ground fatalities the routes have to take into account populated areas and avoid them if possible in case the aircraft is not able to stay airborne during the trajectory to the reachable airport.

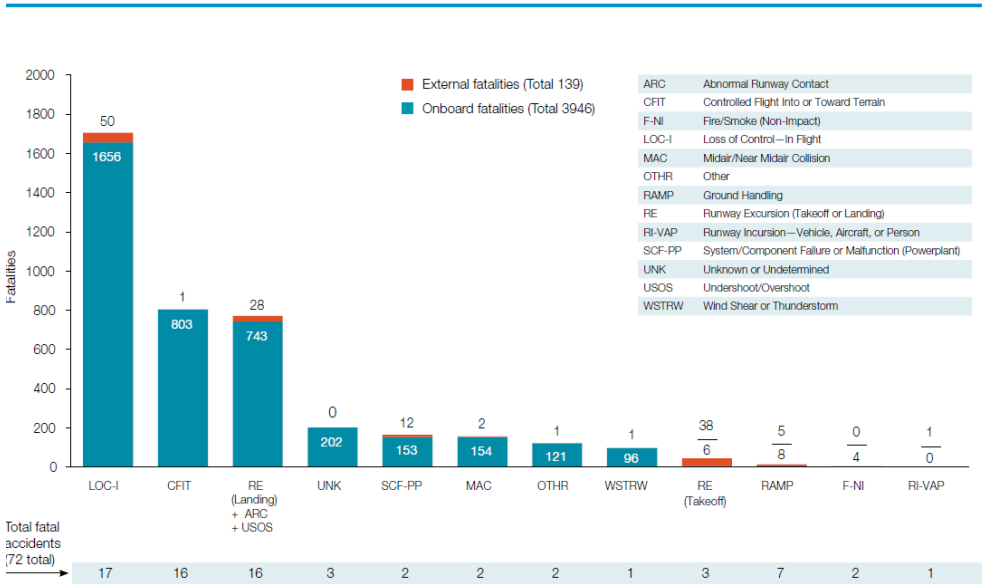


Figure 1.1: Fatal accidents worlwide commercial jet fleet 2005 through 2014 (Boeing, 2014)

1.2. Literature

To date several different systems have been proposed for transport aircraft as well as Unmanned Areal Vehicles (Boskovic and Mehra, 2003; Coombes et al., 2013). Several models have been developed for situations in which an aircraft suffers from engine failure immediately after taking of (Rogers, 1995; Jett, 1982; Hoffren and Raivio, 2000; Brinkman and Visser, 2007). Others for aircraft that suffer actuator failure (Strube et al., 2004). This research focuses on loss of thrust during cruise and assumes that all actuators function normally.

The study done by Chen and Pritchett (2001) focuses on the pilot interaction with a system that generates landing paths. Results of landings in emergency situations with and without the help of a path generating system were compared. It was concluded that a system that generates paths, has a positive influence on the landing procedure and that lack off notifying the pilot of all trade offs can create an over reliance or under reliance of the system. Atkins et al. (2006) developed a framework based on the system of Chen and Pritchett (2001). First the system determines the maximum range the aircraft is able to fly when experiencing full loss of thrust taking into account a degrading aircraft system. This so called *footprint*, which can be seen in figure 1.2, is defined as a circle in which the radius is determined by the best glide path straight ahead and two symmetrical boundary points 120° on either side of the circle marked with 'x'. The aircraft in this system assumes a best glide bank of 30°.

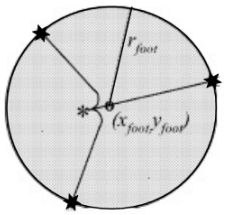
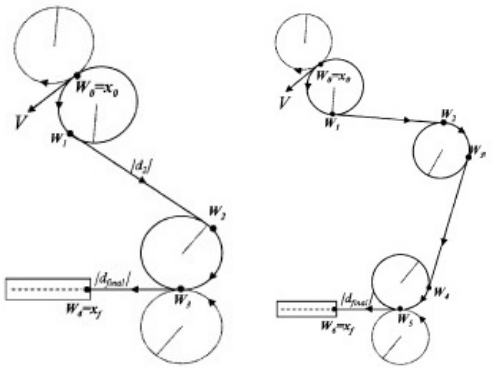


Figure 1.2: Generated footprint (Atkins et al., 2006)

Within the footprint the system proposed by Atkins et al. (2006) tries to construct a Dubins path of guaranteed minimum length to each airport. In figure 1.3 it can be seen that, depending on the situation, the path can be changed and/or stretched. The paths are created by using user defined waypoints $W_m = \{x_m, y_m, h_m, \psi_m\}$ where x_m and y_m indicate the horizontal position, h_m the altitude and ψ the heading angle. If for example the descent angle γ in figure 1.3a is too steep between waypoint W_1 and W_2, the path is stretched by adding an extra turn enabling the aircraft to descent with a reduced descent angle resulting in the path shown in figure 1.3b. A created path is considered feasible if the flight paths angles between the waypoint satisfy the constraint $\gamma \in [\gamma_{min}, \gamma_{max}]$ where γ_{min} is the 'steepest allowed descent angle' and γ_{max} the 'best-glide descent'.



(a) (b)

Figure 1.3: Dubins path (Atkins et al., 2006)

If multiple routes are possible, they are ranked using a safety oriented utility function U (Atkins et al., 2006) defined as:

$$U = \sum_i C_i \cdot w_i = C_1 \cdot \frac{r_1}{r_{1,max}} + C_2 \cdot \frac{r_w}{r_{w,max}} + C_3 \cdot q_I + C_4 \cdot (\frac{d}{d_{max}}) + C_5 \cdot \frac{w_h}{w_{h,max}} + C_6 \cdot \frac{(w_{c,max} - w_c)}{(w_{c,max} - w_{c,min})} + C_7 \cdot q_s + C_8 \cdot q_f, \tag{1.1}$$

where r_l is the runway length, runway width r_w, instrument approach quality q_I, distance d from the footprint boundary, headwind velocity w_h, crosswind velocity w_c, surface quality q_s and facility availability measure q_f.The weighting factors have to be determined by the respective experts, this could be airlines, air traffic controllers or pilots. The function U could help solve the earlier mentioned problem of Chen and Pritchett (2001) concerning "over and under reliance" because it gives the pilot control to choose by providing a list of all reachable airports.

Coombes et al. (2013) use a more advanced footprint in their research. As can be seen in figure 1.4, the footprint boundary is limited by the aircraft movements. If the aircraft makes a turn, the speed has to be increased to maintain vertical equilibrium, which decreases the maximum glide range. For the landing procedure, Coombes et al. (2013) use a so called 'high key low key' technique in which an aircraft is required to be at a specific altitude at a certain location. This procedure is shown in figure 1.5. The five points on the clockwise circuit (cDW,cEB1,cEB2,cDH1,DH2) or the anti-clockwise circuit descent path (aDW,aEB1,aEB2,aDH1,DH2) have to be calculated in order to make a feasible approach route.

Peng et al. (2011) adopt the framework provided by Atkins et al. (2006) and use preset dynamic pressure profiles to generate trajectories. By assuming monotonous decreasing altitude of the aircraft and substituting flight velocity with dynamic pressure enables to use preset dynamic pressure profiles to determine maximum range. This maximum range depends on the initial speed and height. This is similar to the energy profile but dependent on altitude instead of range to go.

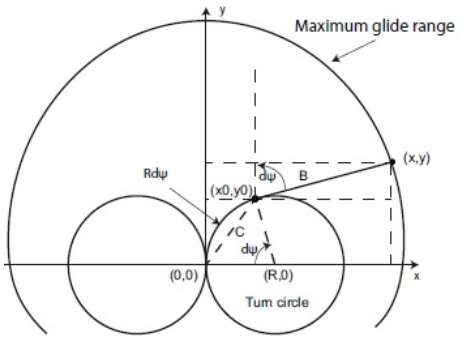


Figure 1.4: Footprint (Coombes et al., 2013)

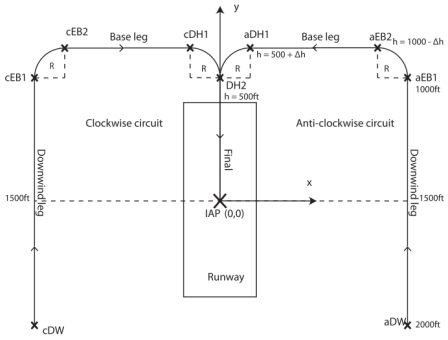


Figure 1.5: The high key low key technique (Coombes et al., 2013)

The models proposed by Meuleau et al. (2009) and Fernandes de Oliveira and Büskens (2013) optimize emergency trajectories to minimize risk of life. The work of Meuleau et al. (2009) takes into account different categories of obstacles, among these are terrain, urban development and weather radar observations. With a visibility graph routes are made using an A* algorithm which finds a sequence of hybrid (discrete and continuous component) states leading from the current state until the chosen runway taking in to account these obstacles that are represented as 2D polygons with a ceiling and floor. Depending on the obstacle (soft or hard), it can be avoided by flying over, alongside and, in the case of the soft obstacles, sometimes through the obstacle. The calculated risk influences the ranking of possible routes to airports generated by the A* algorithm.

The model of Fernandes de Oliveira and Büskens (2013) adopts the risk based approach of Meuleau et al. (2009) with determining the risks on-board and on the ground and generating routes minimizing the probability of loss of life. Fernandes de Oliveira and Büskens (2013) note that the use of an A* based approach does not fully take into account aircraft dynamics and eventual limitations related to the emergency situation. Although the method is determined to be computational efficient, Fernandes de Oliveira and Büskens (2013) conclude that the generated A* trajectories have limited quality. To resolve this quality issue, Fernandes de Oliveira and Büskens (2013) define the risks in an optimal control problem, optimizing trajectories to avoid hazardous weather and flying over populated areas taking into account aircraft dynamics and the aircrafts flight envelope. However the direct approach used by Fernandes de Oliveira and Büskens (2013) has the disadvantage that it is computationally expensive (Betts, 1998; Garg, 2011).

The system of Wu et al. (2012); Wu and Mora-Camino (2013) uses dynamic programming to train a neural network to optimize only the feasible vertical glide trajectory towards a safe landing place. Routes are generated to the runways available in the footprint. Actuator use is minimized by penalizing the use of it, resulting in a smooth gliding path without unnecessary use of the remaining hydraulic energy. The neural network is trained by considering different variables and parameters such as altitude, speed, glide path angle and distance to the landing site. In emergency situations, the trained network generates paths based on the information in its database by giving pitch angle directives. Direct application to on-line gliding control is considered infeasible

due to the involved computational burden. To alleviate this burden, the amount of available states is reduced. This is achieved by using a heuristic melting procedure where states are clustered to create a central state.

The high computational burden of using Dynamic Programming is also underlined by Adler et al. (2012). In their research it is stated that, Dynamic Programming can assure globally optimal solutions via an exhaustive search, however it becomes impractical for problems with more than a few state dimensions.

In the field of robotics Dynamic Programming (DP) is used for path planning (Sallaberger, 1995; Shin and McKay, 1984; Kala et al., 2012), however Sallaberger (1995) like Atkins et al. (2006) and Wu et al. (2012) criticize the algorithm to be 'computationally very expensive'. Powell (2007) suggests Approximate Dynamic Programming (ADP) as a solution for this curse. Szepesvari (2009) states that ADP can turn the infeasible Dynamic Programming methods into practical algorithms so that they can be applied to large-scale problems. Goswami et al. (2010); Konar et al.; Viet et al. (2011); Macek et al.; Aranibar and Alsina (2004) use ADP for path planning in robotics while Santos et al. (2012) use ADP to find the shortest paths in simulated games. To find the optimal path, depending on the situation a different ADP algorithm is used. To create trajectories in an unknown environment, the algorithm simulates paths and receives rewards for reaching the goal and penalties for passing through obstacles. The rewards and penalties earned by visiting a certain state is called the 'value' and is stored in a value function, in which for every state the value is defined. The paths are continued to be simulated until the value function is converged. Assuming a minimization problem, by passing through the states with the lowest values from the start to the goal state the optimal path for the value function is found.

1.3. Research Definition

1.3.1. Objectives

The objective of this research is:

To develop an emergency trajectory planner that can generate trajectories to landing sites in situations with total loss of thrust taking into account loss of life.

In order to achieve the objective the following goals have been set:

1. To create a model that generates feasible gliding trajectories in emergency situations to land the aircraft safely in a short amount of time.
2. To create an Approximate Dynamic Programming algorithm that generates trajectories in an acceptable short time to be able to implement it on-line.
3. To design a model that takes into account passenger and on ground risk when generating emergency trajectories.

1.3.2. Research Questions

Based on the objective and goals, the following research question and sub-question are formulated:

1. How can an emergency trajectory planner generate gliding trajectories using Approximate Dynamic Programming to safely land a transport aircraft with total loss of thrust?
    1.1. What methods are currently available to maximize glide range for aircraft with full loss of thrust?
    1.2. How is the safety of the passengers on board and people on the ground assured while following the emergency trajectory?
    1.3. Which simulation/optimization models have been used for this problem?

1.4. Report Structure

In chapter 2 the architecture of the emergency trajectory planner is presented and discussed. The functions of the individual elements of the trajectory planner are elaborated and it is explained how the elements are connected. In chapter 3 the Approximate Dynamic Programming (ADP) algorithm that is used to generate the trajectories for the emergency planner is extensively discussed. Firstly the theory of Dynamic Programming is discussed and explained with examples. With the concept of Dynamic Programming introduced, the ADP algorithm is explained thoroughly. After the theory, the discussed variants of ADP are compared in two scenarios with a benchmark to determine which algorithm performs the best. The algorithm that performs the best is chosen to create trajectories in the model and further discussed in chapter 4. Chapter 5 presents the results of the research. The trajectories created will be displayed and analyzed. Finally the conclusions of this study based on the results of chapter 5 as well as the recommendations will be presented in chapter 6.

2

The Trajectory Planner

The planning of the trajectories in emergency situations is done in several phases by several modules as can be seen in figure 2.1. The planner proposed in this research is based on the Adaptive Flight Planner of Atkins et al. (2006). In this chapter the system architecture shown in figure 2.1 is explained beginning with the first module, in which the footprint is generated.

2.1. Footprint

The footprint is an approximation of the region in which feasible solutions can be found. In this footprint every runway is considered reachable, however this does not assure a feasible trajectory. The footprint depends on the altitude, speed and aircraft type. In this research it is assumed that the speed of the aircraft at the start of the glide path is equal to the optimal glide speed calculated in equation 2.3. The specifications of the Boeing 737-300 model that is used in this research can be found in table 2.1. The footprint is defined as a circle with a radius determined by the maximum horizontal gliding distance. The circle is not adjusted to account for the aircraft maneuvering like in the research of Coombes et al. (2013). To determine the maximum horizontal gliding distance, first the maximum glide path has to be calculated which follows from the minimum glide angle. The angle is achieved with the maximum lift-to-drag ratio assuming a parabolic drag polar and is calculated with the following equation (Vinh, 1993):

$$(\gamma_d)_{min} = 2\sqrt{KC_{D_0}}, \tag{2.1}$$

where K is the induced drag factor, C_{D_0} the zero-lift drag coefficient $C_d = C_{d_0} + KC_L^2$ and $\gamma_d = -\gamma$.

The angle is assumed to be small so that:

$$\cos\gamma_d \approx 1 \tag{2.2}$$

The speed V (m/s) to fly while following $(\gamma_d)_{min}$ is:

$$V = \sqrt{\frac{2W}{\rho S}\sqrt{\frac{K}{C_{D_0}}}}, \tag{2.3}$$

where W is the aircraft weight (N), ρ the air density (kg/m^3) and S the aircraft wing surface (m^2).

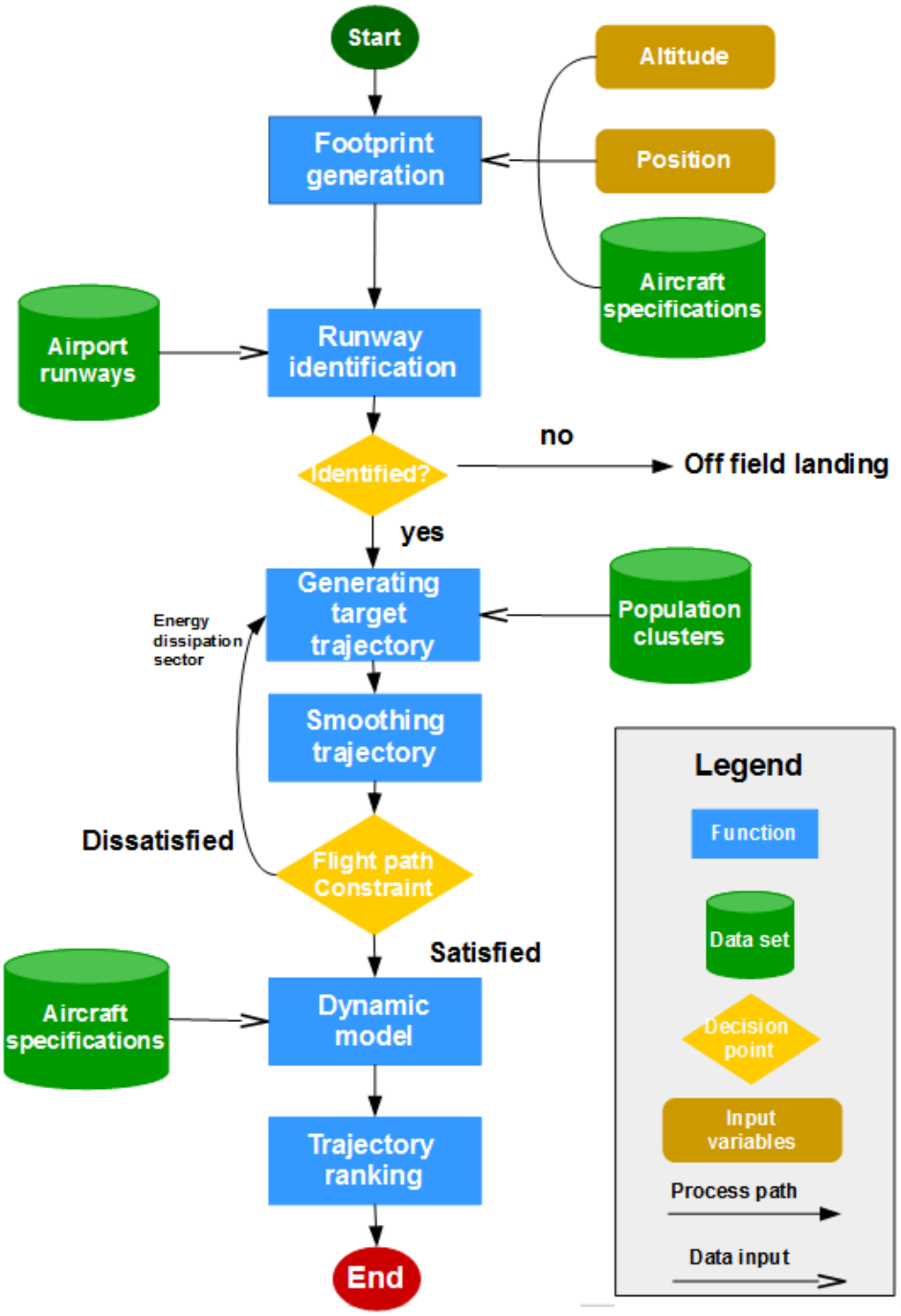


Figure 2.1: Architecture of the trajectory planner

Parameter	Value
Wing surface (S)	105.4 m^2
Weight (W)	550000 N
Induced drag parameter (K)	0.0425
Zero-lift drag coefficient (C_{D_0})	0.0210
Maximum lift coefficient ($C_{L_{max}}$)	1.4

Table 2.1: Boeing 737 model (clean configuration)

To estimate the maximum glide range, the following equations are used:

$$V_{vert} = V \cdot \sin(\gamma_d) \tag{2.4}$$
$$V_{hor} = V \cdot \cos(\gamma_d) \tag{2.5}$$
$$\Delta t = \frac{d_{vert}}{V_{vert}} \tag{2.6}$$
$$r = \Delta t \cdot V_{hor}, \tag{2.7}$$

where d_{vert} is the vertical distance and r the radius of the footprint which is equal to the maximum glide range.

Combining equation (2.1) and equation 2.3, V_{vert} also called the *rate of descent* can be calculated which is shown in equation 2.4. The horizontal distance is defined by equation 2.5. Using a constant descent angle $(\gamma_d)_{min}$ and knowing the vertical distance, the distance between the initial altitude and the runway altitude assumed to be at 0 m, a rough estimate of the time of descent in straight flight can be made which can be seen in equation 2.6. In equation 2.7 the radius of the footprint is determined by multiplying the calculated time of equation 2.6 and the initial horizontal speed of equation 2.5. It is assumed that the aircraft is at cruise altitude when the emergency starts. Using the data from table 2.1 the radius of the footprint is calculated to be 167 km for an initial altitude of 10 km.

An example of a footprint is shown in figure 2.2 in which the initial aircraft position, the possible airports and the footprint are indicated. The detailed map of The Netherlands is created by using data points of CBS (2015) that define province borders with the dutch system of coordinates, the so called Rijksdriehoekscoördinaten (RD-coordinates). The axis of the map are normalized such that a square is formed around the Netherlands with point (0,0) in the map equal to RD-coordinates (4393,306505) with scale 1:100.

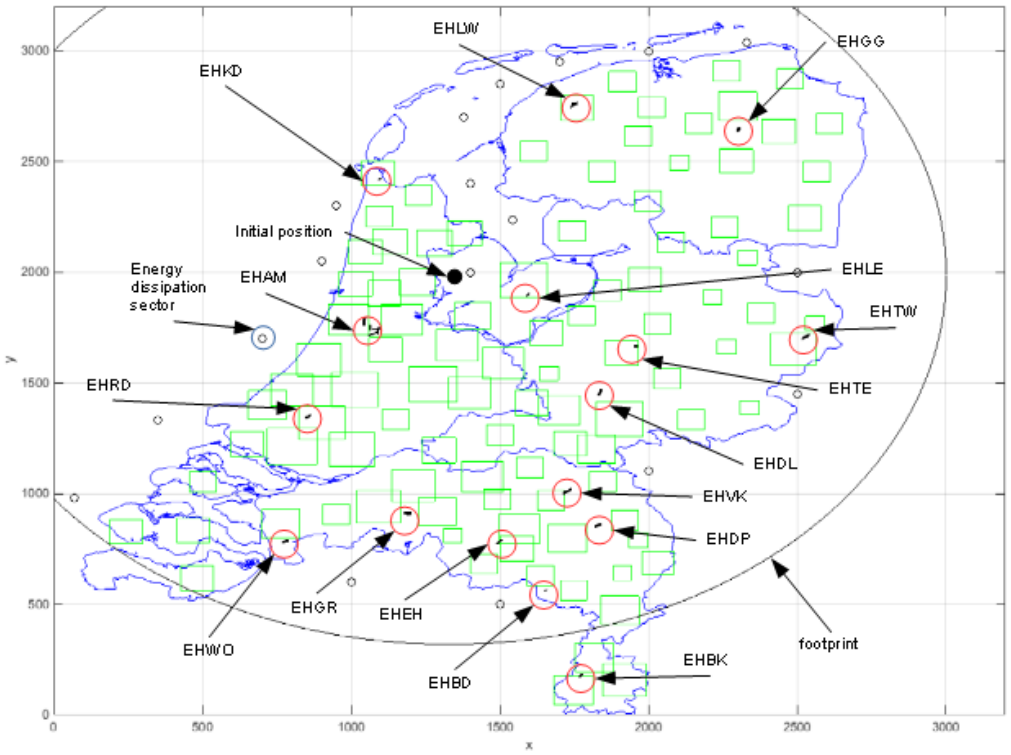


Figure 2.2: Map with footprint

2.2. Trajectory Generation

If there are runways found in the footprint, the next step is to create trajectories to the runways. To achieve the objective of the research, stated in section 1.3.1, the trajectory planner has to take into account risk while planning the routes. Section 1.2 briefly mentioned models in which trajectories are optimized to minimize risk of life which is achieved by taking into account several safety constraints. The first constraint taken into account is *population avoidance*.

2.2.1. Population Avoidance

Avoiding flying over populated areas when in a state of emergency reduces the risk of on ground casualties. To evaluate this risk a similar approach as employed by Fernandes de Oliveira and Büskens (2013) has been used. The risks associated to the generated trajectories are taking into account population data of The Netherlands provided by CBS. The dataset which has a resolution of 0.01 km^2 contains information regarding population distribution in the whole area of The Netherlands.

The population of a country is spread all over the surface of the country. Some regions are highly populated while other areas have a low population density. If each house has to be avoided it would be impossible to generate trajectories. To overcome this problem the population of the Netherlands is clustered into 100 groups. The approach chosen to achieve this the *K-means* method is used.

To start, the K-means algorithm needs three parameters to be specified by the user:

1. Number of clusters.
2. Initialization of cluster centroids.
3. Distance metric, typically Euclidean.

Considering that the population data is represented by set X= $\{x_i\}$, $i = 1,..,n$ and is clustered into a set of K clusters, $C = \{c_k, k = 1,....,K\}$. The position of the clusters are determined by minimizing the distance between the mean of the cluster μ_k and the points in the cluster c_k, the so called *squared error* (Jain, 2010):

$$J(c_k) \sum_{x_i \in c_k} \| x_i - \mu_k \|^2 \tag{2.8}$$

The sum of the squared error is then minimized over all K clusters:

$$J(c_k) \sum_{k=1}^{K} \sum_{x_i \in c_k} \| x_i - \mu_k \|^2 \tag{2.9}$$

Minimizing equation 2.9 is an iterative process to accurately determine the cluster centers with the following main steps (Jain, 2010):

1. Select an initial partition with K clusters; repeat steps 2 and 3 until cluster membership stabilizes.
2. Generate a new partition by assigning each pattern to its closest cluster center.
3. Compute new cluster centers.

This iterative process is depicted in figure 2.3.

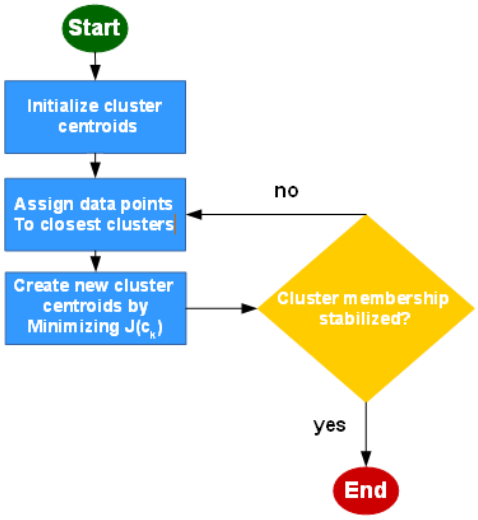


Figure 2.3: K-means

With the position of the 100 clusters known, the dimensions of the areas have to be determined which depend on the average distance from the cluster center to the data points pertaining to that cluster and the population density of the cluster. For each cluster the distances of each data point to the center are aggregated in the variable *sumd*. Dividing *sumd* by the number of data points, that is the inhabitants, gives the average distance which is taken as the distance from the center of the square to one of the four vertices and is portrayed as the black line in figure 2.4.

If the dimensions would only be based on the average distance, this would mean that in rural zones with low population density, the average distance would create huge areas which do not represent correctly the actual population. To solve this problem, it has been assumed that in each cluster the population density is at least 1000 inhabitants/ km^2. To achieve this, the surfaces of big areas in rural zones are adjusted (shrunk) until this condition is met. To show the effect of the area adjustment an example of an area with low population density is illustrated in figure 2.5a. It can be seen that the adjusted area (red square) is smaller than the original area (blue square). The (brown) circle in the middle of the area indicates the cluster center.

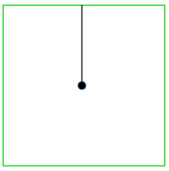


Figure 2.4: Obstacle defining

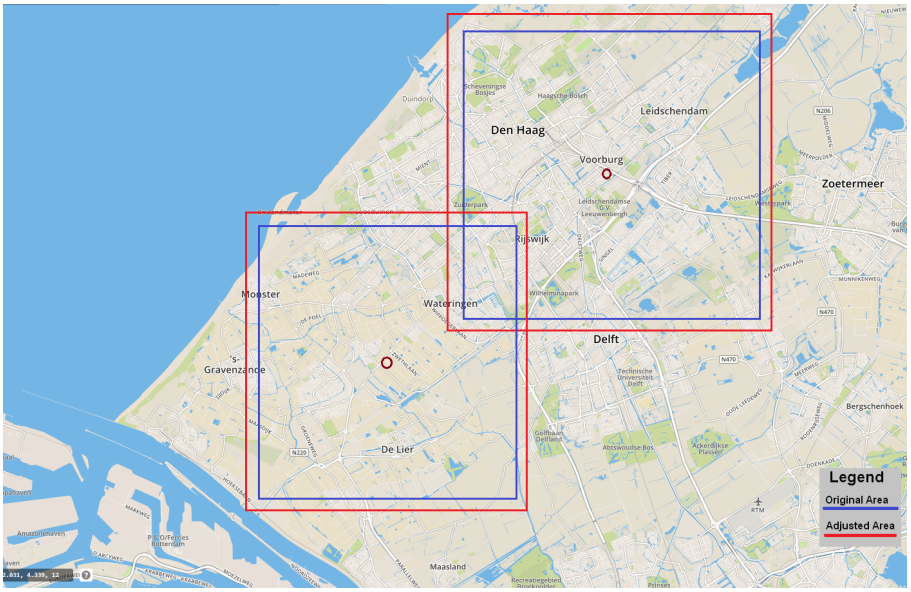
For the areas with more than 1000 inhabitants/ km^2 the dimensions are also adjusted. The factors that increase the surface of the areas are educated guesses and are listed in table 2.2.

Table 2.2: Population density factors

Population Density(PD)/km^2	Factor
$1000 < PD < 2000$	1.05
$2000 < PD < 3000$	1.10
$3000 < PD < 4000$	1.15
$PD > 4000$	1.2

The fact that an area is more densely populated does not increase an area, however flying over this area has a higher potential risk and should therefore be avoided. By increasing the area the chance that a trajectory is created over densely populated zones is decreased. The differences between the original area and the adjusted area in a high population density area in figure 2.5b.

(a) Low population density



(b) High population density

Figure 2.5: Comparing original and adjusted area

2.2.2. Horizontal Trajectories

The horizontal trajectories are called *target trajectories* because they serve as a reference for the next phase, *Dynamic Model* in which routes are generated in the 3D plane, meaning horizontal as well as vertical. In the previous section it was noted that by avoiding population, the risk of on ground casualties is reduced, however according to Meuleau et al. (2009) the risk of diminishing controllability of the aircraft increases with time and distance. The generated trajectories must therefore by an equilibrium of population avoidance and shortest path. The target trajectories are created by an algorithm known as Approximate Dynamic Programming (ADP) and will be explained extensively in chapter 3. The ADP algorithm does not take into account heading angles when generating the target trajectories which can result in routes in which the aircraft is not aligned with the runway. To cope with this, the trajectories are generated to a *target point* at a distance indicated in figure 2.6 from the threshold of the runway aligned with the orientation of the runway. It is assumed that the distance is sufficient to correct the possible angle difference between the heading angle of the aircraft and the runway orientation. The horizontal distance between the runway threshold and the *target point* results from the altitude of 305 meters (≈ 1000 ft) at an angle of 3 degrees at which the aircraft should be stabilized when flying on the glide slope.

For each created trajectory it will be checked if the descent angle satisfies the constraint $\gamma_d \in [(\gamma_d)_{min}, (\gamma_d)_{max}]$ where $(\gamma_d)_{min}$ is the best glide descent and $(\gamma_d)_{max}$ the steepest allowed descent angle. The aircraft is assumed to be in steady gliding flight if the descent angle falls in this range. To determine if the descent angle satisfies the constraint, the *smoothing trajectory* function shown in the system architecture in figure 2.1 estimates the ground track of the 3D trajectory by creating a route using the dynamic model, which will be explained in section 2.3.

The length of the ground track is an estimation because the ground track is slightly different depending on the flight path angle. For the estimation, the best glide descent angle is used. With the estimated ground track and initial altitude it is calculated if the needed flight path angle, to descent to the altitude of the target point indicated in figure 2.6, satisfies the constraint. Using equation 2.1 and the aircraft specifications from table 2.1 the value of best glide descent $(\gamma_d)_{min}$ is calculated to be 3.4° and the value of $(\gamma_d)_{max}$ is set equal to 6°. If the flight path angle needed for the trajectory is too steep, the trajectory will not be feasible and an alternative trajectory will be generated featuring a stretched path. This alternative trajectory consists out of two segments starting with the segment from the initial position to an area called *Energy Dissipation Sector*. The second part of the trajectory is generated between the energy dissipation sector and the reachable airport.

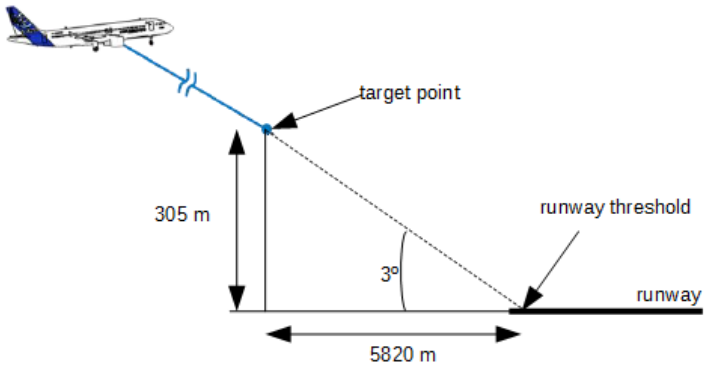


Figure 2.6: Adapted from Fernandes de Oliveira and Büskens (2013)

2.2.3. Energy Dissipation Sector

In these sectors possible excess speed and altitude can be dissipated by using the speed brake and gradually descending by spiraling down. The sectors are placed in uninhabited zones and are indicated in figure 2.2. In the figure it can be seen that next to the sectors placed in the North Sea and Wadden Sea, also deviation sectors are placed on the border with Belgium and Germany in areas which are assumed to be uninhabited.

2.3. Dynamic Model

When the target trajectories are created, these are only generated in the horizontal plane and do not take into account aircraft maneuver limitations. A point mass trajectory dynamics model is used to create a smoothed trajectory in the horizontal plane as well as in the vertical plane using the earlier created target trajectories as a reference path. The dynamic model used in this research is based on the dynamic module made in Matlab by Lazos Fernandez (2015). The nonlinear equations of motions of the aircraft are numerically integrated using a Runge-Kutta integration and minimize errors by using a time step of 0.2 seconds. The equations describing the point mass trajectory dynamics in the 3D space are (Vinh, 1993):

$$\dot{X} = V\cos\gamma\ cos\psi \tag{2.10}$$

$$\dot{Y} = V\cos\gamma\ sin\psi \tag{2.11}$$

$$\dot{Z} = V\sin\gamma \tag{2.12}$$

$$\dot{V} = \frac{-Dg}{W} - g\sin\gamma \tag{2.13}$$

$$\dot{\psi} = \frac{g}{V}\tan\mu \tag{2.14}$$

$$\dot{\gamma} = \frac{g}{V}\Big(\frac{L}{W}\cos\mu - \cos\gamma\Big), \tag{2.15}$$

where X, Y, Z denote the aircraft position in the inertial reference frame, g the acceleration of gravity (m/s^2), μ is the bank angle and ψ is the heading angle. The drag and lift forces are denoted by respectively D and L.

Drag force D is defined by:

$$D = C_{D_0} qS + \Big(\frac{L}{W}\Big)^2 K\frac{W^2}{qS}, \tag{2.16}$$

where q is the dynamic pressure.

The control variables in this model are bank angle μ and flight path angle γ. Assuming equilibrium of forces in the vertical plane ($\dot{\gamma}$=0), equation 2.16 can be rewritten to an equation in which the direct influence of the bank angle on the drag is seen:

$$D = C_{D_0} qS + (1+\tan^2(\mu))K\frac{W^2}{qS} \tag{2.17}$$

The model described above embodies the following assumptions:

- Standard atmosphere conditions
- Flat, non-rotating earth
- Wind is neglected
- Speed loss due to ram air turbine is neglected
- Coordinated turns (no side slip)

The parameters of the aircraft model and *International Standard Atmosphere* (ISA) values used to calculate the values of the model can be found in appendix A.3.

A lateral autopilot is used to control the heading of the aircraft. It does this by selecting the bank angle such that the heading follows a first order response:

$$\dot{\psi} = \frac{1}{\tau_{\psi}}(\psi_c - \psi) \tag{2.18}$$

where ψ_c is the commanded value of the heading angle and τ_{ψ} the gain of the response of the autopilot to a command (Lazos Fernandez, 2015). The commanded heading angle follows from the target trajectory generated by the ADP algorithm. The positions of the smoothed trajectory, approach the positions of the target trajectory. At each position of the smoothed trajectory it is determined which angle must be chosen as the commanded heading angle to approach the next position of the target trajectory.

It is required to also calculate the associated bank angle. As can be seen in equation 2.17 the bank angle influences the drag directly and thereby affecting the glide range. The value of μ is determined by:

$$\mu = \arctan\Big(\frac{V\dot{\psi}}{g}\Big), \tag{2.19}$$

where $\dot{\psi}$ is evaluated using equation 2.18. It is noted that the bank angle is constrained within a range [μ_{min}, μ_{max}].

As explained in section 2.2.2, to determine if a direct route is possible from the current aircraft position to the runway, the flight path angle has to be determined. First an estimation of the ground track is made using the dynamic model with descent angle $(\gamma_d)_{min}$. With the vertical distance, the difference between the altitude at the initial aircraft position and the altitude at the target point, and the estimated ground track known, the descent angle for the trajectory is determined. If this descent angle satisfies the constraint $\gamma_d \in [(\gamma_d)_{min}, (\gamma_d)_{max}]$ a direct trajectory is possible. This direct trajectory made by the dynamic model will consists out of one segment and will have a constant descent angle. Using a loop for the dynamic model, the exact descent angle for the trajectory is determined.

If the flight path angle is to steep and does not satisfy the constraint, the trajectory will comprise two segments. The first segment will be generated between the current aircraft position and the energy dissipation sector and the second segment between the energy dissipation sector and the runway. The flight path angle is constant for both parts of the route and equal to $(\gamma_d)_{min}$.
The second part of the trajectory from the energy dissipation sector to the airport, will be generated backwards by the dynamic model from the target point until the dissipation sector. Because the model stops the trajectory before the landing sequence starts, it will not take into account the use of flaps and landing gear. This terminal point is defined in figure 2.6. The second part of the trajectory is generated backwards because the conditions at the target point are fixed but the conditions at the energy dissipation sector are variable in order to create a feasible route. A route is considered to be feasible (physically possible) if the altitude of the trajectory from the start to the energy dissipation sector is higher than the altitude of the backward integrated trajectory from the airport to the energy dissipation sector.

In the last phase of the trajectory planner, the feasible trajectories are ranked.

2.4. Trajectory Ranking

In section 2.2.1 the first safety constraint, *population avoidance*, was introduced. This safety constraint is used to create trajectories avoiding population. The multiple generated routes reach different airports which vary in size and available facilities. The emergency facilities available at an airport influence the risk of landing at that specific airport. If in an emergency situation the aircraft is uncontrollable when it touches the ground and the facilities at the airport are not sufficient, it can have a large impact on the airport safety. By ranking the possible airports this risk can be reduced.

To achieve this the utility based prioritization function U (Atkins et al., 2006) already introduced in section 1.2 is used, however because in this research wind is not taken into account, the function used is slightly adjusted resulting in the following equation:

$$U = \sum_i C_i \cdot w_i = C_1 \cdot \frac{r_1}{r_{1,max}} + C_2 \cdot \frac{r_w}{r_{w,max}} + C_3 \cdot q_I + C_4 \cdot (\frac{d}{d_{max}}) + C_5 \cdot q_s + C_6 \cdot q_f, \tag{2.20}$$

where r_l is the runway length, r_w is the runway width, q_I instrument approach quality, distance d from the footprint boundary, surface quality q_s and facility availability measure q_f. The values of r_l, r_w and d are normalized [0.0 1.0]. The weighting factors have to be determined by the respective experts; this could be airlines, air traffic controllers or pilots. In this research the weighting factors determined by Atkins et al. (2006) are used which are set to $\{C_1, C_2, C_3, C_4, C_5, C_6\}$= $\{0.15, 0.15, 0.15, 0.15, 0.1, 0.1\}$.

The value of U determines the rank of each created trajectory. As stated earlier the values of r_l, r_w and d are normalized and are different for each initial condition. The values of q_I, q_s are fixed and depend on the facilities at the airport. In this research the values of q_I, q_s determined by Atkins et al. (2006) are used. These are listed in figure 2.7.

Description (max value used)	Value
Instrument approach (q_I)	
WAAS, ILS/MLS	1.0
LOC with RWY designation	0.95
LOC w/o RWY designation	0.85
LDA w/RWY designation	0.8
LDA w/o RWY designation	0.7
GPS, LORAN, RNAV w/RWY	0.6
VOR, NDB, SDF w/RWY	0.5
GPS, LORAN, RNAV, VOR, NDB, SDF w/o RWY	0.2
Runway surface (q_S)	
Asphalt	1.0
Concrete	1.0
Metal, brick, etc. (VTOL)	0.5
Wood	0.2
Turf/gravel/dirt	0.1
Airport facilities (q_f)	
Fuel of required type (Jet-A)	0.25
Airframe maintenance	
Major	0.25
Minor	0.125
Power plant maint.	
Major	0.25
Minor	0.125
Bulk oxygen	0.25

WAAS = wide area augmentation system, ILS = instrument landing system, MLS = microwave landing system, LOC = localizer, RWY = runway, LDA = localizer type directional aid, GPS = global positioning system, LORAN = long range navigation, RNAV = area navigation, VOR = very high frequency omni-directional range, NDB = nondirectional beacon, SDF = simplified directional facility, and VTOL = vertical takeoff and landing.

Figure 2.7: Quality measures for runway utility computation (Atkins et al., 2006)

The primary risk factors according to Meuleau et al. (2009) are *runway length, width* and *relative wind.* The authors assume as a general rule 12 meters of runway for each knot of speed at touchdown. This risk could be reduced by only taking into account runways that surpass the minimum runway length needed by the aircraft. According to Brady (2015) the minimum runway length for a Boeing 737-300 is 1400 meters. In emergency situations however a runway must be found to get the aircraft on the ground. To increase the chance of finding a runway, the minimum runway length has been lowered to 1200 meters. The shorter runway will certainly affect the ranking of the airport but provides more possible solutions. The airports in The Netherlands that satisfy the conditions are listed in Appendix A.2 and are indicated in figure 2.2.

3

Dynamic Programming

In chapter 1 it could be seen that several different approaches have been used to generate trajectories in emergency situations. One of these approaches was Dynamic Programming, which was deemed effective but problematic to use on board in problems with large dimensions due to the so called 'curse of dimensionality'. This curse refers to the fact that computational requirements grow exponentially with the number of state variables (Sutton and Barto, 1998). In this chapter Dynamic Programming and a solution for the curse will be discussed in order to create the target trajectories.

3.1. Dynamic Programming

Dynamic Programming (DP) is a term defined by Richard Bellman to describe a class of methods that can be used to solve problems that are complex and large, by breaking it down into subproblems and finding the optimal solutions for the subproblems. If these subproblems are overlapping, then the optimal solutions of the subproblems will result in a global optimum. According to Sutton and Barto (1998) Dynamic programming refers to 'a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov Decision Process (MDP)'. This is also underlined by Busoniu et al. (2010b) who states that the DP problems can be formalized as a MDP which is defined by Kallenberg (2007) as a 'model for sequential decision making under uncertainty, taking into account both the short-term outcomes of current decisions and opportunities for making decisions in the future'.

If the decision of taking a certain action only depends on the present state and action and is not influenced by previous decisions it is said it has the *Markov Property*. If the MDP model is known, an optimal solution can be found using Dynamic Programming. If the dimensions are too large, using DP to find the optimal solution becomes unfeasible and a method called Approximate Dynamic Programming (ADP) might be used. These two methods to solve MDP problems will be presented and explained in this chapter.

3.1.1. Markov Decision Process

With the *action* and *state* space known, the MDP is defined by the transition probability function $\mathbb{P}$ and its reward function $\mathbb{R}$. The transition probability function is defined as $\mathbb{P}(s' = s_{t+1}|s = s_t, a = a_t)$ and determines the probability of reaching state s' when taking action a in state s. The expected value of the reward is defined as $\mathbb{R}= \mathbb{E}(s'|s,a)$. A reward is defined by Sutton and Barto (1998) as 'the intrinsic desirability of that state' and will be further explained in section 3.1.2.

The MDP is usually used for stochastic problems, but can also be used for deterministic problems (Busoniu et al., 2010b). In this research only deterministic problems are assumed. For deterministic MDP the transition probability function is defined as $\mathbb{P}(s'|s,a) = 1$ and the reward function $\mathbb{R}= \mathbb{E}(s'|s,a) = 1$ (Ortner, 2010).

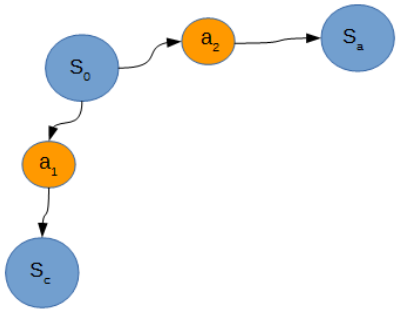


Figure 3.1: Deterministic Markov Decision Process

Using the example in figure 3.1 this would mean that when in state S_0 action a_1 is chosen, the next state will be S_c. The same holds for the situation in which action a_2 is chosen, this will lead to state S_a.

3.1.2. Value Functions

The optimal solution for a MDP is found by passing through the states with the lowest values (in this research minimization problems are assumed). To determine the optimal solution, the optimal *policy* has to be found which leads to the states with the lowest values. Policy is defined as the mapping for actions for each state, in other words it determines which action a is chosen in each state s. To find the optimal policy, first the value function for policy π has to be found. This can be calculated using the recursive Bellman equation:

$$V^{\pi}(s_t) = r_{t+1} + V^{\pi}(s_{t+1}), \tag{3.1}$$

where r_{t+1} is the reward 'earned' by going from state s_t to state s_{t+1} and $V^{\pi}(s_{t+1})$ the value of the next state. The rewards in equation 3.1 can be divided into two categories, 'the immediate reward' (r_{t+1}) and the 'longterm reward' (V_{t+1}). The immediate reward is earned by going from the current state s_t to the next state s_{t+1} while the longterm reward $V(s_{t+1})$ shows the rewards that might be earned from state s_{t+1} until the final state summed up. The final state of the solution, also called the *terminal state*, is always zero. This can be explained by the fact that in the last state no rewards for reaching the final state can be received because it is already reached. V^{π} is called the *state-value* function for policy π.

In problems concerning for example finance, assuming inflation, in which the value of currency reduces over time, a discount rate is used. Discount rate γ ($0 \leq \gamma \leq 1$) determines the present value of future rewards. This means that future rewards have to be discounted based on the fact that they will not be earned directly but in the future.

Another reason to discount according to Busoniu et al. (2010a) is to ensure bounded rewards even with an infinite horizon. In the following chapters, the discount factor γ will be used to formulate generic equations for all types of problems, finite and infinite. If γ is zero, the decision which state will be visited next is only based on minimizing the immediate reward r_{t+1}, and will not take into account the possible future rewards $V(s_{t+1})$. A policy that only takes into account immediate rewards is called *myopic*. When γ is equal to 1, future earned rewards are not discounted. The general version of the Bellman's equation:

$$V^{\pi}(s_t) = r_{t+1} + \gamma V^{\pi}(s_{t+1}) \tag{3.2}$$

The value function can also be determined separately for all possible actions in each state. For each policy this can be determined and is called the *action-value* function Q^{π}. It is defined as:

$$Q^{\pi}(s_t, a_t) = r_{t+1} + Q^{\pi}(s_{t+1}, a_{t+1}), \tag{3.3}$$

where $Q^{\pi}(s_{t+1}, a_{t+1})$ is the value of Q in the next state by taking action a_{t+1} determined by the policy.

The optimal value function results from the optimal policy and is defined as:

$$V^*(s) = \min_\pi V^\pi(s), \tag{3.4}$$

for all s $\in \mathbb{S}$

The optimal action-value function has the same policy as the optimal state-value function and is defined as:

$$Q^*(s) = \min_\pi Q^\pi(s,a) \tag{3.5}$$

for all s $\in \mathbb{S}$ and $a \in \mathbb{A}(s)$.
Equations 3.4 and 3.5 determine respectively the optimal state-value function and action-value function by minimizing over all different functions determined by different policies. V^* and Q^* are correlated in such a way that the action a that minimizes value $Q^*(s,a)$ of state s is equal to $V^*(s)$. Consequently V^* can be calculated by:

$$V^*(s) = \min_{a_t \in A_t} Q^{\pi^*}(s,a) \tag{3.6}$$

$$V^*(s) = \min\Big\{ r_{t+1} + \gamma\, V^*(s_{t+1}) | s_t = s, a_t = a \Big\} \tag{3.7}$$

This is called Bellman's optimality equation.

Bellman's optimality equation for the action-value function is defined as:

$$Q^*(s,a) = r_{t+1} + \gamma \min_{a_t \in A_t} Q^*(s_{t+1}, a_{t+1}) \tag{3.8}$$

3.1.3. Policy Iteration

To find the optimal policy, several methods can be used. One of these methods is *policy iteration*. It consists out of two steps respectively the *policy evaluation* followed by *policy improvement*.

Policy Evaluation

Starting with the policy evaluation, the state-value function V^π of policy π is determined iteratively for each state:

$$V_{i+1}(s_t) = r_{t+1} + \gamma V_i(s_{t+1}), \tag{3.9}$$

where i is the number of iterations. The stopping condition for the evaluation process is:

$$max_{s \in \mathbb{S}}(|V_i - V_{i+1}|) < \Delta \tag{3.10}$$

The policy can also be evaluated for the action-value function Q^π:

$$Q_{i+1}(s_t, a_t) = r_{t+1} + \gamma Q_i(s_{t+1}, a_{t+1}) \tag{3.11}$$

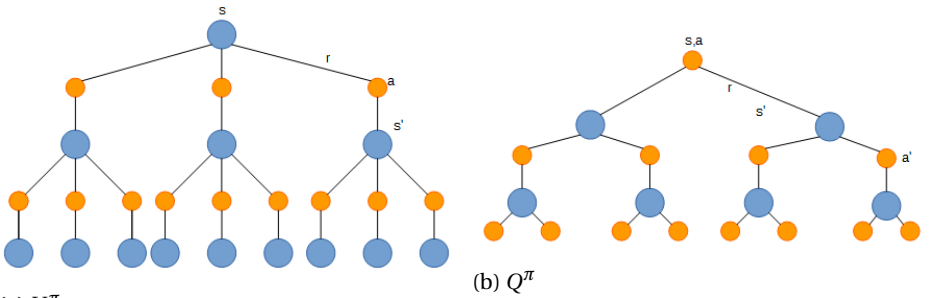
With the equivalent stopping condition:

$$max_{s \in \mathbb{S}}(|Q_i - Q_{i+1}|) < \Delta, \tag{3.12}$$

where Δ is a to be defined low value.

It can be seen that equations 3.9 and 3.11 are the Bellman equation for respectively the state-value function (equation 3.2) and action-value function (equation 3.3). After each iteration the stopping condition is tested. For $i \to \infty$ the value function in equation 3.9 will converge to V^* (Sutton and Barto, 1998). With the value function iterated for the policy, the next step is to try to find a policy better than the current one.

In figure 3.2 the backup diagrams for the state-value function V^π and the action-value function Q^π are shown. The difference between the two is that while the backup diagram in figure 3.2a is made for state s, the backup diagram in figure 3.2b is made for state action pair s, a.



(a) V^π (b) Q^π

Figure 3.2: Backup diagrams Policy Iteration (Sutton and Barto, 1998)

Policy Improvement

With *policy improvement*, for all the states it is checked which action in each separate state leads to the best next state. This is also called *greedy policy improvement*. For the state-value function this is determined by:

$$\pi'(s_t) = arg\min_a(r_{t+1} + \gamma V^\pi(s_{t+1})) \tag{3.13}$$

The greedy policy improvement for the action-value function:

$$\pi'(s_t) = arg\min_a Q^\pi(s_t, a_t) \tag{3.14}$$

The value function of the new policy π' is evaluated using policy evaluation until the stopping conditions is satisfied (equation 3.10 and 3.12). The policy iteration continues until there is no policy better found than the current policy. Algorithm 1 shows the Policy Iteration process in pseudocode.

With policy iteration, a potential bad policy is evaluated multiple times until iteration stops. Sutton and Barto (1998) state that policy iteration can be 'truncated without losing the convergence guarantee', resulting in a method called *value iteration*. Powell (2007) notes that it might be the most widely used algorithm in dynamic programming because it is the simplest to implement.

3.1.4. Value Iteration

With value iteration all states are backed up once after which the policy in all the states is improved. The state-value function is approximated iteratively for each state:

$$V_{i+1}(s_t) = \min_a(r_{t+1} + \gamma V_i(s_{t+1})), \tag{3.15}$$

for all $s \in \mathbb{S}$.

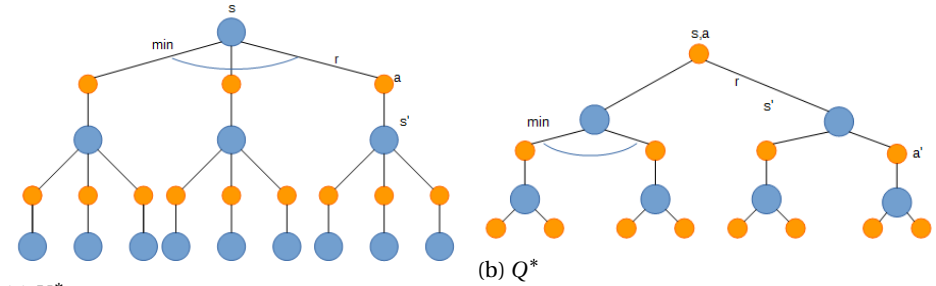
Value iteration for the action-value function can be calculated with:

$$Q_{i+1}(s_t, a_t) = r_{t+1} + \gamma \min_{a_t \in A_t} Q_i(s_{t+1}, a_{t+1}) \tag{3.16}$$

Algorithm 1: Policy iteration (Sutton and Barto, 1998)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathbb{A}$ arbitrarily for all $s \in \mathbb{S}$
2. Policy evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathbb{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow r_{t+1} + \gamma V_i(s_{t+1})$
   $\Delta \leftarrow max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)
3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathbb{S}$
   $b \leftarrow \pi(s)$
   $\pi'(s_t) \leftarrow arg\min_a(r_{t+1} + \gamma V^{\pi}(s_{t+1}))$
   if $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ false
   If *policy-stable*, then stop; else go to 2

The value iteration stops when there is no policy better than the current one. The backup diagrams for the value iteration process are displayed in figure 3.3. It can be seen that the backup diagrams of policy iteration and value iteration are almost identical except for the minimization of the choices of respectively states in figure 3.3a and action in figure 3.3b. This can be explained by the fact that while value iteration and policy iteration use Bellman's equation, as noticed before, value iteration uses Bellman's optimality equation to iteratively compute an optimal value function, from which an optimal policy is derived (Busoniu et al., 2010a). Algorithm 2 shows pseudocode for Value Iteration.



(a) V^* (b) Q^*

Figure 3.3: Backup diagrams Value Iteration (Sutton and Barto, 1998)

Algorithm 2: Value iteration (Sutton and Barto, 1998)

Initialize V arbitrarily, e.g., $V(s) = 0$, for all $s \in \mathbb{S}$

Repeat
$\Delta \leftarrow 0$
For each $s \in \mathbb{S}$:
$v \leftarrow V(s)$
$V(s) \leftarrow \min_a(r_{t+1} + \gamma V(s_{t+1}))$
$\Delta \leftarrow max_{s \in \mathbb{S}}(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π , such that

$\pi(s) = arg\ \min_a(r_{t+1} + \gamma V(s_{t+1}))$

3.1.5. Example

In sections 3.1.3 & 3.1.4 the theories of *Policy Iteration* and *Value Iteration* have been explained. To better understand these methods, the following example is used.

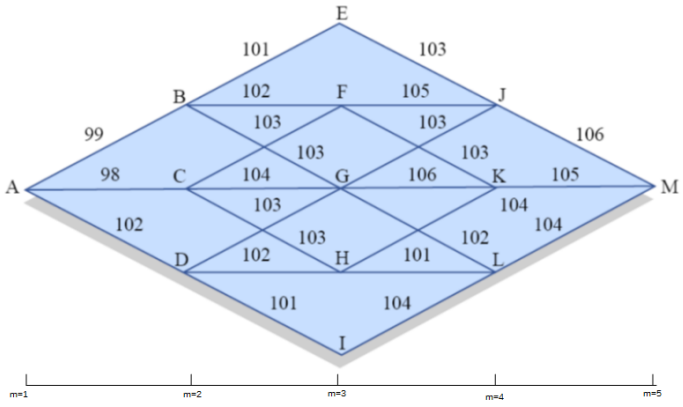


Figure 3.4: Example 1 (Visser, 2015)

In this minimization problem, displayed in figure 3.4 the optimal route between state A and state M has to be found. At each stage m, there are multiple possible next states. In stage $m = 1$ for example the next state will be B, C, D. The optimal route will go through the states with the lowest values and therefore is the route with the lowest summed up rewards. The rewards, this could be the time or fuel consumption between two states, are the black numbers and the value of each state will be displayed in green. It is assumed that $V_0 = 0$ and that the problem will not be discounted ($\gamma = 1$).

The Policy Iteration and Value Iteration algorithms start in stage m=1 and end in stage m=5 and are used to find the optimal solution shown in figure 3.5.

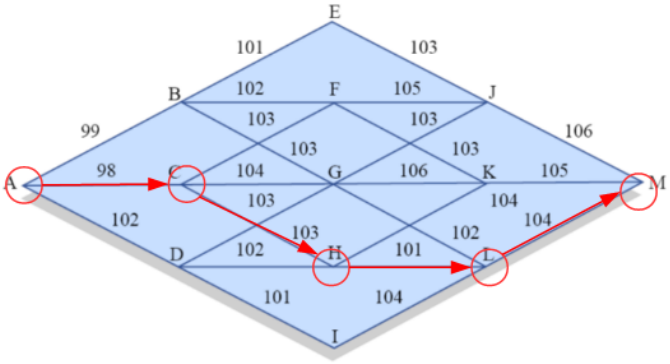


Figure 3.5: Optimal solution example 1

Policy iteration

As earlier explained for a given policy the values of all states are iterated until it converges to V^{π}. In this example the iteration starts with policy π_1 for which the next state in every state is indicated with the black arrows in figure 3.6a.

The values for all states will be calculated using the iterative update version of Bellman's equation:

$$V_{i+1}(s) = r_{t+1} + \gamma V_i(s_{t+1}) \tag{3.17}$$

It is assumed that $V_0 = 0$ for all states. The resulting values for all states are shown in table 3.1. To clarify how these values are determined the values for states A, B, C and D for the first iteration are calculated as follows:

$V_1(A) = 102 + V_0(D) = 102 + 0 = 102$
$V_1(B) = 103 + V_0(G) = 103 + 0 = 103$
$V_1(C) = 103 + V_0(H) = 103 + 0 = 103$
$V_1(D) = 101 + V_0(I) = 101 + 0 = 101$

To calculate the values of A, B, C and D in the second iteration the values of their following states, determined by policy π_1, are needed. These values can be found in table 3.1. The values of A, B, C and D in the second iteration will be:

$V_2(A) = 102 + V_1(D) = 102 + 101 = 203$
$V_2(B) = 103 + V_1(G) = 103 + 102 = 205$
$V_2(C) = 103 + V_1(H) = 103 + 101 = 204$
$V_2(D) = 101 + V_1(I) = 101 + 104 = 205$

	States													
Iteration	A	B	C	D	E	F	G	H	I	J	K	L	M	Policy
1	102	103	103	101	103	103	102	101	104	106	105	104	0	π_1
2	203	205	204	205	209	208	206	205	208	106	105	104	0	
3	307	309	308	309	209	208	206	205	208	106	105	104	0	
4	411	309	308	309	209	208	206	205	208	106	105	104	0	
5	411	309	308	309	209	208	206	205	208	106	105	104	0	
6	406	309	308	307	209	208	206	205	208	106	105	104	0	π_2
7	406	309	308	307	209	208	206	205	208	106	105	104	0	

Table 3.1: State values Policy Iteration

This iteration process will be repeated for all states until the maximum difference between the value of all states of the current iteration and the previous iteration is smaller than $\Delta = 0.1$:

$$max_{s \in \mathbb{S}}(|V_i - V_{i+1}|) < \Delta \tag{3.18}$$

As can be seen in table 3.1 the difference between the values of the fourth and fifth iteration satisfy the stopping conditions and therefore stop the evaluation phase. The values of all states of evaluated policy π_1 can be seen in figure 3.6b displayed in green.

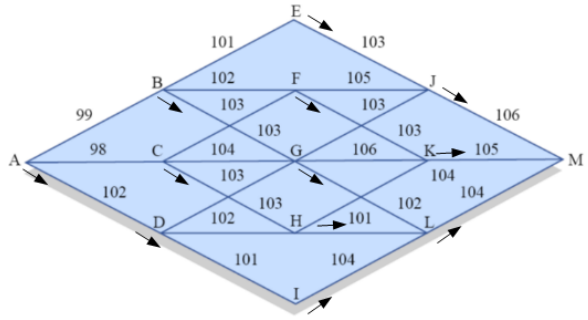
With the value function for π_1 known, the next step is to determine if the current policy is optimal for all states:

$$\pi'(s) = arg\min_{a}(r_{t+1} + \gamma V^{\pi}(s_{t+1})) \tag{3.19}$$

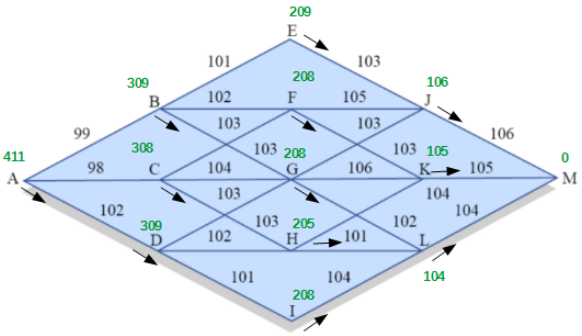
With the current policy π_1 after state A, D is visited. Using equation 3.19 at state A, it is determined if the current policy is optimal:

$arg\min_{A \to B}(99 + 309) = 408$
$arg\min_{A \to C}(98 + 308) = 406$
$arg\min_{A \to D}(102 + 309) = 411$

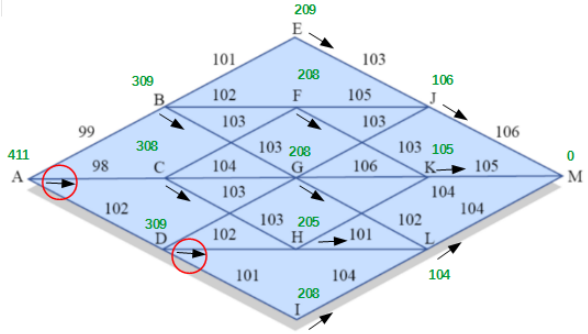
From these calculations it can be concluded that a better solution is found by changing the policy in state A to visit state C next instead of state D. These calculations are made for all states and result in the new policy, π_2, shown in figure 3.6c. The changes between policy π_1 and π_2 are indicated with the red circles in figure 3.6c. The new policy is also evaluated. The resulting values for all states are shown in table 3.1 and in green in figure 3.6d. In turn policy π_2 is also tested for improvement, however there is no policy found that is better than π_2. The resulting path follows the policy indicated with arrows in figure 3.7 and is equal to the optimal solution shown in figure 3.5.



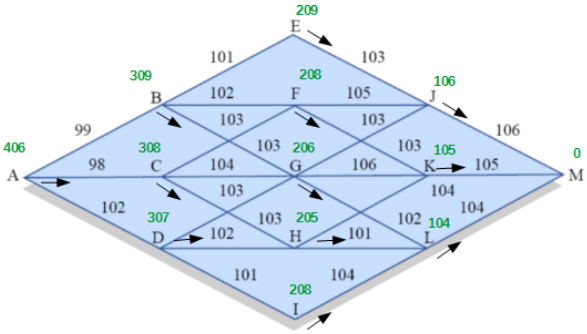
(a) π_1



(b) π_1



(c) π_2



(d) π_2

Figure 3.6: Policy Iteration

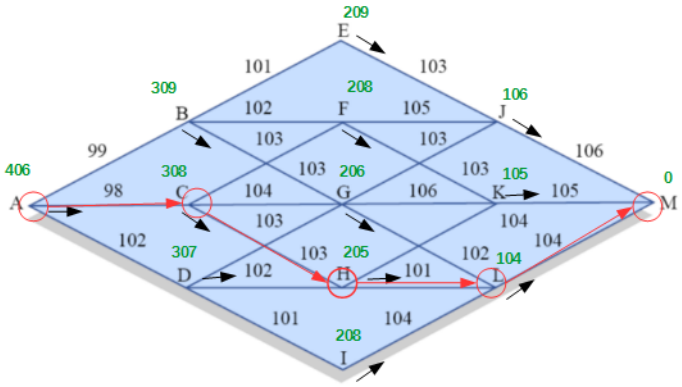


Figure 3.7: Policy Iteration optimal solution example 1

Value Iteration

It is recalled that the value iteration algorithm makes a backup of all states and afterwards, if necessary, the policy is improved. In example 1 this is done for all states in each stage m:

$$V_{i+1}(s) = \min_a(r_{t+1} + \gamma V_i(s_{t+1})) \tag{3.20}$$

The values for all states for the several needed iterations are shown in table 3.2. To clarify how value iteration calculates the values of states, the first iteration for states A,C and G is shown.

$$V_1(A) = \min\Big\{99 + V_0(B)\Big|98 + V_0(C)\Big|102 + V_0(D)\Big\}$$
$$V_1(A) = \min\Big\{99 + 0\Big|98 + 0\Big|102 + 0\Big\}$$
$$V_1(A) = 98$$

$$V_1(C) = \min\Big\{103 + V_0(F)\Big|104 + V_0(G)\Big|103 + V_0(H)\Big\}$$
$$V_1(C) = \min\Big\{103 + 0\Big|104 + 0\Big|103 + 0\Big\}$$
$$V_1(C) = 103$$

$$V_1(G) = \min\Big\{103 + V_0(J)\Big|106 + V_0(K)\Big|102 + V_0(L)\Big\}$$
$$V_1(G) = \min\Big\{103 + 0\Big|106 + 0\Big|102 + 0\Big\}$$
$$V_1(G) = 102$$

The policy changes after each iteration and is displayed with black arrows for each iteration in figure 3.8. In the calculations can be seen that the lowest value for A is achieved by visiting state C after A. For state C there are two states that give the same value. In this example the chosen next state is determined to be F, the other possible option is indicated with the transparent arrow. The changing policy for each iteration is shown in figure 3.8. The changes between policies determined by the above explained calculations are indicated with red circles. After iteration III the policy does not change anymore. The stopping conditions is achieved after the fourth iteration as can be deduced from the values in table 3.2. The optimal solution found with value iteration is displayed in figure 3.8d and is equal to the optimal solution found by the policy iteration.

	States												
Iteration	A	B	C	D	E	F	G	H	I	J	K	L	M
1	98	101	104	101	103	103	102	101	104	106	105	104	0
2	200	204	204	203	209	208	206	205	208	106	105	104	0
3	302	309	308	307	209	208	206	205	208	106	105	104	0
4	406	309	308	309	209	208	206	205	208	106	105	104	0
5	406	309	308	309	209	208	206	205	208	106	105	104	0

Table 3.2: State values Value Iteration

In this chapter it was explained how Value and Policy iteration work. In each state the state value function $V(s)$ was determined, the value however can also be calculated for each action in a state separately resulting in the action-value function $Q(s,a)$. In the example in this chapter, the values of all states have to be backed up. The computational power needed to calculate this increases exponentially with the number of state variables. A simple calculation shows how fast the amount of dimensions increase. Assume that there are $N = 100$ different states in which $L = 4$ different actions can be taken. This small problem already results in $N^L = 1 \cdot 10^8$ possible outcomes.
To overcome this curse, Approximate Dynamic Programming (ADP) has been presented as a solution (Powell, 2007) and this will be explained in the next section.

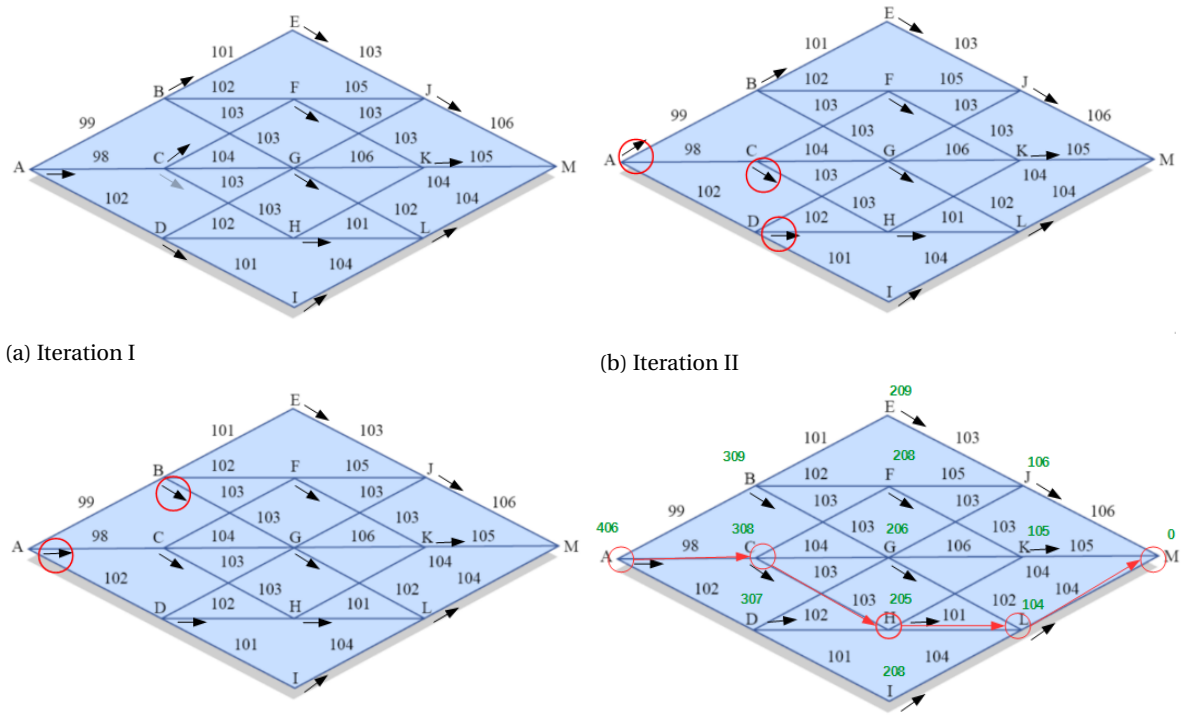


Figure 3.8: Value Iteration

3.2. Approximate Dynamic Programming

According to Sutton and Barto (1998) another term for ADP is 'Reinforcement Learning' (RL). Gosavi (2014) states that 'the power of Reinforcement Learning lies in its ability to solve, near-optimally, complex and large-scale MDP's on which classical DP breaks down'. Szepesvari (2009) underlines the use of RL to 'turn the infeasible Dynamic Programming methods into practical algorithms so that they can be applied to large-scale problems'. As explained in section 3.1.3 the first step in policy iteration is, policy evaluation. Kunz (2013) states that one of the most used approaches for policy evaluation in Reinforcement Learning is *Temporal Difference (TD) Learning*. This method will be explained in the following section.

3.2.1. Temporal Difference Learning

TD calculates the value function, or the approximation of the value function, by minimizing the error of the temporal consecutive predictions. By taking k steps following a certain policy π, rewards are received and *backed up* determining a new estimate. The difference between the new estimate and the old estimate is called the *temporal difference error* and is used to update the value of the state. This is repeated for all the states which are visited by the 'agent' of the TD algorithm by following policy π until the goal is reached. The sequence of steps from the start state until the goal, is called *episode* or *trial* and is repeated until the value function is determined.

The values of the states $V(s_t)$ are calculated incrementally (Sutton and Barto, 1998):

$$\bar{V}(s_t) \leftarrow \bar{V}(s_t) + \alpha\delta_t, \tag{3.21}$$

where $\bar{V}(s_t)$ is the estimation of the value in state s_t, α the *step size* and δ_t the *temporal difference error*. Here the '$\leftarrow$' notation means that the right hand side of the equation updates the left hand side incrementally.

Temporal difference error δ_t is defined by:

$$\delta_t = R_t^{(k)} - \bar{V}_t(s_t), \tag{3.22}$$

where the return R_t is defined as:

$$R_t^{(k)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{k-1} r_{t+k} + \gamma^k V_t(s_{t+k}) \tag{3.23}$$

Combining equation 3.21 and equation 3.22 results in the detailed TD equation:

$$\bar{V}(s_t) \leftarrow \bar{V}(s_t) + \alpha[R_t^{(k)} - \bar{V}_t(s_t)] \tag{3.24}$$

There are two different ways of updating the values of the states. The *on-line update* is done during the episode as soon as error δ_t is calculated:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha\delta_t \tag{3.25}$$

With *off-line* updating, temporal difference error δ_t is not used to update the value of the states until the end of the episode is reached:

$$V(s_t) + \sum_{t=0}^{T-1} \delta_t \tag{3.26}$$

The range of k steps is between 1 and the full return as can be seen in figure 3.9.

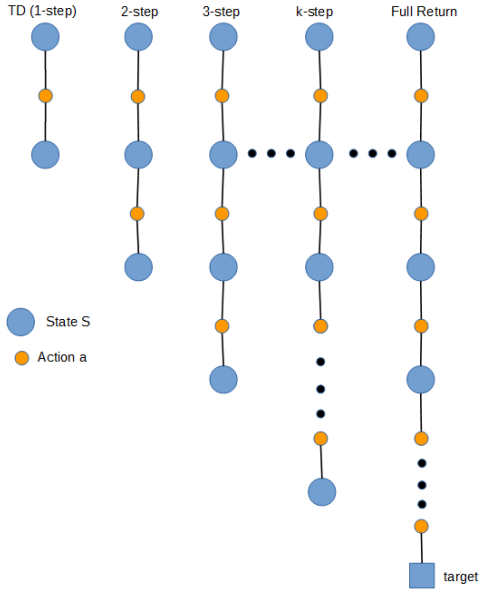


Figure 3.9: K-steps return (Sutton and Barto, 1998)

The 1-step TD is also referred to as 'the simplest TD method' and is one of the two extremes of the k-steps return. The return for the 1-step method is calculated using equation 3.23 for k=1:

$$R_t^{(1)} = r_{t+1} + \gamma \bar{V}_t(s_{t+1}) \tag{3.27}$$

The TD error for the 1-step method using equation 3.22 is:

$$\delta_t = r_{t+1} + \gamma \bar{V}_t(s_{t+1}) - \bar{V}_t(s_t), \tag{3.28}$$

where (r_{t+1}) is the direct reward and V_{t+1} the future rewards. If the method of estimating the value of a state in itself is based on an existing estimate, in this case V_{t+1}, it is said that the method is *bootstrapping*. Algorithm 3 shows pseudocode for tabular TD(0).

In figure 3.10 it is shown how 1-step TD is related to backup diagram of DP. Following policy π a sample is made in contrary to DP where a full backup is done.

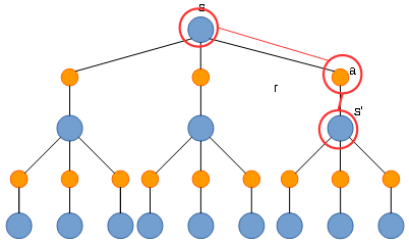


Figure 3.10: Backup diagram TD

Algorithm 3: Tabular TD(0) (Sutton and Barto, 1998)
Initialize $V(s)$ arbitrarily, π to the policy to be evaluated Repeat (for each episode): Initialize s Repeat (for each step of episode): $a \leftarrow$ action given by π for s Take action a; observe reward, r, and next state, s' $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$ $s \leftarrow s'$ until s is terminal

The other extreme of the k-steps return is the full return which is also known as the *Monte Carlo* (MC) method. This term might be confusing because it normally refers to a class of methods involving significant repeated random sampling. Sutton and Barto (1998) however, use this term specifically for methods based on complete returns:

$$R_t = (r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-t-1} r_T), \tag{3.29}$$

where the T is the time step at the terminal state (goal).

If the return from equation 3.29 is filled in equation 3.22 the TD error is:

$$\delta_t = [(r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-t-1} r_T) - \bar{V}_t(s_t)] \tag{3.30}$$

The Monte Carlo method bases the value of the state on the full return of the sample path and therefore has to wait until the end of the episode to increment $\bar{V}(s_t)$. The Monte Carlo method and the 1-step TD can be compared using the example introduced in section 3.1.5. When sampling paths with a greedy policy, in this example two possible paths are possible. This is explained by the fact that at state C, two of the three immediate rewards are equal (103). The path chosen to explain the difference between both methods is chosen to be the path shown in figure 3.11 as the continued red line while the other possible route is indicated with a dotted transparent red line.

Assuming $V^0 = 0$ and $\gamma = 1$ using the 1-step TD method the value of state A after the first iteration will be determined by the temporal difference error δ_t :

$$[(98 + 0) - 0] \tag{3.31}$$

The TD error using the full return backup is given by:

$$[(98 + 103 + 103 + 105) - 0] \tag{3.32}$$

The two backup diagrams are portrayed in figure 3.12.
According to Sutton and Barto (1998) 'on-line and off-line TD prediction methods using k-steps backups converge to the correct predictions under appropriate technical conditions', however the use of the 1-step, k-steps or Monte Carlo backup result in different prediction errors.

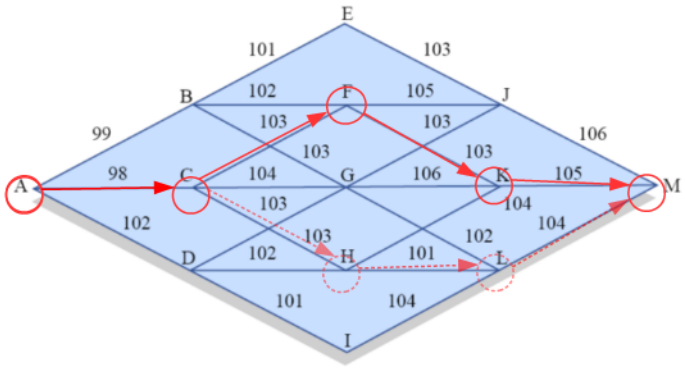


Figure 3.11: TD greedy policy sample path example 1

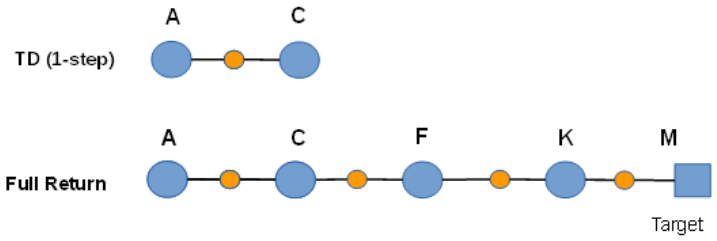
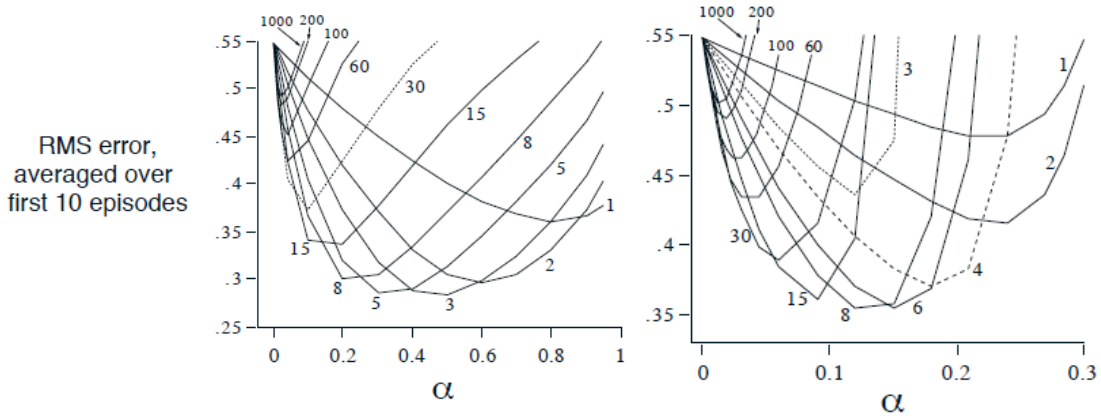


Figure 3.12: TD backup diagrams for example 1

In figure 3.13 an experiment performed by Sutton and Barto (1998) shows the Root Mean Square (RMS) error between the true values of the states and the values found by TD. In this experiment, in which 19 states are visited by a random walk, different k-steps backups are used. It can be seen that the extremes of the k-steps for both on-line and off-line have the biggest (RMS) error. Sutton and Barto (1998) note that this underlines the effectiveness of combining the Monte Carlo Method with Temporal Difference learning. Determining the



(a) On-line k-step TD

(b) Off-line k-step TD

Figure 3.13: Performance k-step TD methods (Sutton and Barto, 1998)

value function of a problem can go on indefinitely, therefore a stopping criterion has to be used. The criterion used by Sutton and Barto (1998) states that iterative policy evaluation should stop if the maximum difference between the new and old estimate of value V is less than Δ:

$$\max_{s \in S} |V_{i+1}(s) - V_i(s)| \leq \Delta \tag{3.33}$$

Note that this is the same stopping criterion as in section 3.1.3.

Exploration versus Exploitation

With TD, a policy is followed to sample paths. If this policy is *greedy*, it means that only states are chosen that give a high immediate reward.By following a greedy policy the same path will be sampled over and over again. For the backup diagram in figure 3.14 this would mean that only the red circled state is visited and the rest of the states that might have higher values are never visited. To overcome this problem the value field should be *explored*, which means that the algorithm should be forced to visit states by taking actions different from the ones determined by the policy. When states are *exploited*, it means that the values of the visited states are used to find the optimal solution. A method often applied to explore uses a so called ϵ-greedy policy (Powell, 2007) in which with probability ϵ at random an action is chosen from the list of possible actions, regardless of the values that can be obtained by taking this action. If an ϵ-greedy policy is used in the backup diagram in figure 3.14 the green circled state can be visited (exploring) while with a greedy policy only the red circles state is visited (exploitation). An equilibrium of both exploration and exploitation has to be found in order to find the optimal solution.

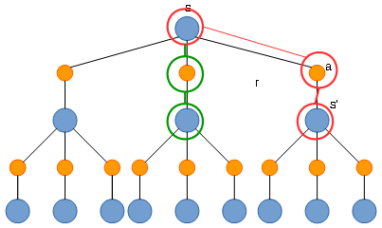


Figure 3.14: Backup diagram TD ϵ greedy

3.2.2. Sarsa & Q-learning

In section 3.1.2 it was shown that for DP the Bellman equation could be used to define the state-value function V and the action-value function Q, this also applies to Temporal Difference learning. Sutton and Barto (1998) identified two main classes with their corresponding methods: *Sarsa* for *on-policy* and *Q-learning* for *off-policy*. An *on-policy* method follows the policy it is learning about while the *off-policy* method can learn from behavior generated by a different policy (Precup et al., 2001).

Sarsa

The name refers to the sequence of calculating the value of states: State, action, reward, (next) state, (next) action (Busoniu et al., 2010a). The values of the 1-step Sarsa are calculated by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3.34}$$

Algorithm 4 shows pseudocode for Sarsa. Note that the ϵ-greedy policy is in respect to the value of Q and not in respect to the immediate reward. According to Sutton and Barto (1998) the convergence depends on 'the nature of the policy's dependence on Q'. If all state-action pairs are visited an infinite amount of times, it will converge to the optimal policy with a probability of 1 (van Seijen et al., 2009).

Algorithm 4: Sarsa (Sutton and Barto, 1998)

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., $\epsilon - greedy$)
  Repeat for (each step of episode):
    Take action a, observe reward, r and next state, s'
    Choose a' from s' using policy derived from Q (e.g., $\epsilon - greedy$)
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'; a \leftarrow a';$
  until s is terminal

Q-learning

Like with Sarsa, the policy determines the states Q-learning will visit. In contrary to Sarsa, Q-learning minimizes over all action-values in the state visited by following the policy where Sarsa only uses the action-value determined by the policy. In order for Q-learning to converge to the optimal policy, it is required that all pairs are updated an infinite amount of times (Sutton and Barto, 1998).

The 1-step Q-learning is determined by:

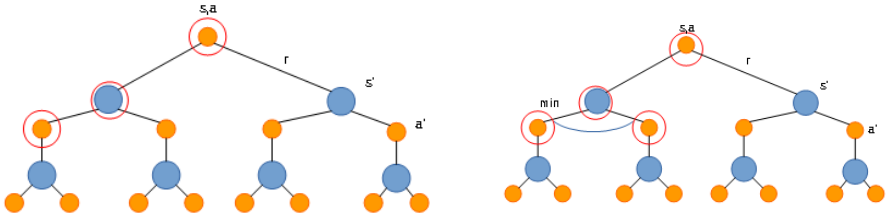
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \min_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3.35}$$

Algorithm 5 shows pseudocode for tabular Q-Learning.

Algorithm 5: Q-Learning (Sutton and Barto, 1998)

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat for (each step of episode):
    Choose a from s using policy derived from Q (e.g., $\epsilon - greedy$)
    Take action a, observe reward, r and next state, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \min_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$;
  until s is terminal

Sarsa and Q-learning are respectively the equivalent of *policy evaluation* and *value iteration* for sample paths. In figures 3.15a & 3.15b the red circle indicate the sample path backup in comparison with the full backup of DP. Although Sarsa and Q-learning look alike, depending on the ϵ, there is a difference in which states are visited. In Sarsa both states (per each Sarsa step) are visited by taking ϵ-*greedy* actions while with Q-learning the first action is ϵ-*greedy* and the next state that will be visited has the minimum value for $Q(s_{t+1}, a_{t+1})$, essentially a *greedy* action. If ϵ is equal to zero, meaning only greedy steps, Sarsa and Q-learning will observe the same rewards and therefore have the same backup (Singh et al., 2000).

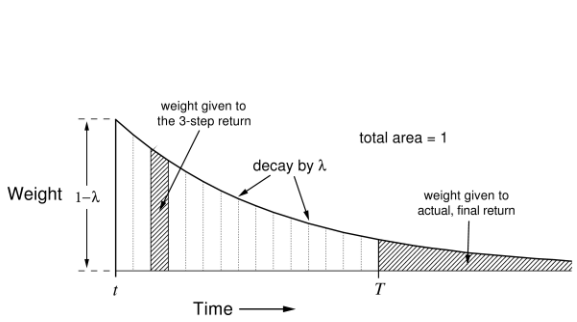


(a) Sarsa (b) Q-learning

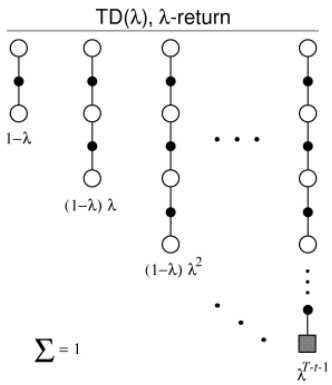
Figure 3.15: Backup diagrams Sarsa & Q-learning (Sutton and Barto, 1998)

3.2.3. TD Lambda

Temporal Difference methods and Monte Carlo both have their advantages and disadvantages. In situations in which episodes are very long, MC is delayed in learning and has to wait until the end of the episode before the return is known while 1-step TD only needs to wait 1 step. Sutton (1988) suggested an algorithm to create 'a bridge from Temporal Difference to Monte Carlo Methods' (Sutton and Barto, 1998), it is called the $TD(\lambda)$ algorithm. Singh and Sutton (1996) labeled it a 'fundamental mechanism to handle delayed reward'. Sutton and Barto (1998) describe two possible views on the TD(λ) namely, the forward view and the backward view.



(a) λ decay (Sutton and Barto, 1998)



(b) λ return (Sutton and Barto, 1998)

Figure 3.16: Forward view

Forward view

The forward view is seen as the 'theoretical' view and is used to explain the more 'practical' backward view. With TD(λ) the more recent earned rewards have a greater effect on the return, meaning that for a k-steps return for every step in the future from the current until k the rewards are weighted with λ. The decay of this weight can be seen in figure 3.16a. The return weighted proportional to λ^{k-1} where $0 \leq \lambda \leq 1$ is called the λ return:

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{k-1} R_t^{(k)}, \tag{3.36}$$

where normalization factor (1-λ) guarantees that the weights sum to 1.

The backup diagrams of the λ return can be seen in figure 3.16b. As might be noticed, it is equivalent to figure 3.9 but with weighted returns. For TD(λ) also holds that the extremes of the range are the 1-step TD, $\lambda = 0$, and the Monte Carlo method, $\lambda = 1$.

With the λ return defined, the increment $\Delta V_t(s_t)$ can be calculated:

$$\Delta V_t(s_t) = \alpha\Big[R_t^\lambda - V_t(s_t)\Big] \tag{3.37}$$

Backward view

In the backward view an additional variable is used, it is called the *eligibility trace*. Every state that is visited is 'marked' by leaving a *trace*. This trace can be seen as a memory of having visited that state. For every time step the eligibility traces of all states decay while the eligibility trace at current state s_t, is incremented with 1. This kind of trace is called *accumulating trace* (Sutton and Barto, 1998):

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$

The accumulation of the eligibility trace due to the times the states are visited is shown in the middle sub-figure 3.17.

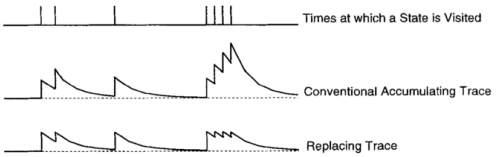


Figure 3.17: Accumulating and replacing traces (Singh and Sutton, 1996)

Taking into account eligibility traces, the temporal difference error is given by:

$$\Delta V_t(s_t) = \alpha \Big[R_t^{\lambda} - V_t(s_t) \Big] e_t(s) \tag{3.38}$$

At each state, depending on the k-steps prediction, the current TD error is propagated back and assigned to the earlier visited states.

With accumulating traces, it can occur that the eligibility trace of certain states acquire large values due to the fact that these are visited much more than others. These states therefore receive a bigger part of future rewards than more recent states. This can result in a biased return and can cause very large (undesirable) updates (Wiering, 1999).

To overcome these problems, Singh and Sutton (1996) suggested *replacing eligibility traces*:

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases}$$

In figure 3.17 can be seen that with replacing traces, every time a state is visited, the trace is reset to 1 while with accumulating traces the value of 1 is added to the current trace. The pseudocode for on-line tabular TD(λ) can be seen in algorithm 6. The value of eligibility trace λ has a big influence on the success of the TD(λ) method.

Algorithm 6: On-line tabular TD(λ) (Sutton and Barto, 1998)

Initialize V (s) arbitrary and e(s)=0, for all s $\in \mathbb{S}$
Repeat (for each episode):
  Initialize s
  Repeat for (each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a, observe reward, r and next state, s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$ (accumulating traces)
    or $e(s) \leftarrow 1$ (replacing traces)
    For all s:
      $V(s) \leftarrow V(s) + \alpha \delta e(s)$
      $e(s) \leftarrow \gamma \lambda e(s)$
    $s \leftarrow s'$
  until s is terminal

For deterministic problems Wiering (1999) suggests large values of λ, so that the trace, updates the values of states for a longer part of the episodes. This however also results in a higher variance in update steps. The value of λ is affected by the value of learning rate α, according to Grzes (2010) for lower values of α higher values of λ are considered better. In figure 3.18 it can be seen that the result of the 19-state random walk experiment, earlier explained in section 3.2.1, performed by Sutton and Barto (1998) underlines the choice of a high λ value for low learning rates. Analyzing figure 3.18 and figure 3.13a, the equivalence of the k-steps backup with a long trace backup for on-line TD can be seen.

3.2.4. Sarsa λ & Q λ

Sarsa (λ)

Eligibility traces can also be applied to Sarsa (Sutton and Barto, 1998), resulting in Sarsa(λ). The Q values are iterated by:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \delta_t e_t(s,a) \tag{3.39}$$

Temporal difference error δ_t is defined as:

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \tag{3.40}$$

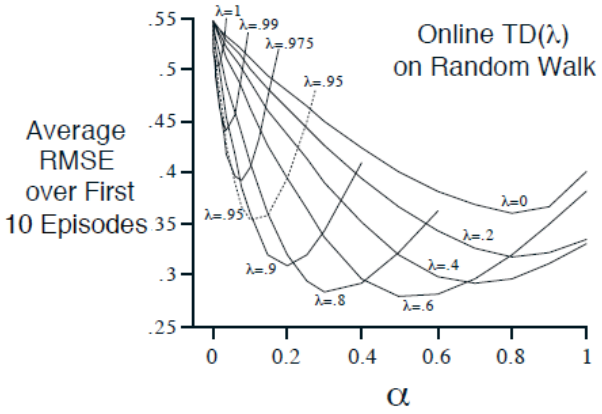


Figure 3.18: Performance on-line TD(λ) (Sutton and Barto, 1998)

The traces are not only dependent on state s but also on action a. For accumulating traces this results in:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & \text{if } s = s_t \; and \; a = a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{if } otherwise \end{cases}$$

Equivalently for replacing traces:

$$e_t(s,a) = \begin{cases} 1 & \text{if } s = s_t \; and \; a = a_t; \\ \gamma\lambda e_{t-1}(s,a) & \text{if } otherwise \end{cases}$$

for all s, a.

As earlier explained in section 3.2.2, Sarsa is an on-policy method meaning that the $Q^\pi(s,a)$ is approximated for the current policy π. Depending on the amount of steps chosen, between 1 and full return, the accumulated rewards are determined and weighted using trace $e_t(s,a)$. The pseudocode for this method is shown in algorithm 7.

Algorithm 7: Tabular Sarsa (λ) (Sutton and Barto, 1998)

Initialize $Q(s,a)$ arbitrary and $e(s,a) = 0$, for all s, a
Repeat (for each episode):
  Initialize s, a
  Repeat for (each step of episode):
    Take action a, observe reward, r and next state, s'
    Choose a' from s' using policy derived from Q (e.g., ϵ-$greedy$)
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$ (accumulating traces)
    or $e(s,a) \leftarrow 1$ (replacing traces)
    For all s,a:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal

Q(λ)

When eligibility traces are applied to Q-learning it is called Q(λ). The method was first introduced by Watkins (1989) and only leaves a trace at the states it visits by taking greedy steps. The trace of the current state-action pair is incremented with 1 while the traces of the other states decay with $\gamma\lambda e_{t-1}(s,a)$.
The first non-greedy (exploratory) step ends the episode and its trace is set to 0:

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1_{s=s_t} \cdot 1_{a=a_t} & \text{if } Q_{t-1}(s_t,a_t) = max_a Q_{t-1}(s_t,a) \\ 1_{s=s_t} \cdot 1_{a=a_t} & \text{otherwise} \end{cases}$$

The algorithm updates the values of Q in the same way as Sarsa(λ)

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t e_t(s,a), \tag{3.41}$$

where

$$\delta_t = r_{t+1} + \gamma \min_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t). \tag{3.42}$$

3.2.5. Learning Rate

Dabney (2014) states that 'the choice of a learning-rate, can profoundly affect the performance of the learning algorithm', it is therefore important to choose an adequate learning rate. For TD learning every state has multiple next states, resulting in multiple possible values for the state. In order to converge the value of $V(s)$ a variable learning rate has to be used to average the values.

According to Powell (2007) learning rate α has to meet the following three conditions:

$$\alpha_{n-1} \geq 0, n = 1, 2, ..., \tag{3.43}$$

$$\sum_{n=1}^{\infty} \alpha_{n-1} = \infty, \tag{3.44}$$

$$\sum_{n=1}^{\infty} (\alpha_{n-1})^2 < \infty, \tag{3.45}$$

where n is the amount of times a certain state is visited.
Choosing the value of the learning rate is important and depends on the problem itself. Sutton and Barto (1998) suggest the following learning rate which meets the three convergence conditions:

$$\alpha_{n-1} = \frac{1}{n} \tag{3.46}$$

This learning rate however puts the highest weight on the early iterations when the estimates are the worst (Powell, 2007). An alternative learning rate that also satisfies the conditions for convergence is proposed by Powell (2007). This learning rate is a generalization of $\frac{1}{n}$:

$$\alpha_{n-1} = \frac{a}{a+n-1}, \tag{3.47}$$

where a is a tuning variable.

Gosavi (2014) proposes a variation on the learning rate in equation 3.47:

$$\alpha_{n-1} = \frac{a}{b+n}, \tag{3.48}$$

where a and b are tuning variables.

With Q-learning and Sarsa, for every state the value $Q(s,a)$ is determined for all actions in that state separately. In a deterministic state space every action taken in a state leads to one specific next state every time. Because the values of each state is determined for every action, in contrary to TD the values do not have to be averaged over all actions per state. Q-learning (Koenig and Simmons, 1996) and Sarsa (van Seijen et al., 2009) can therefore use learning rate $\alpha = 1$. Assuming however that an ϵ-greedy policy is used, a small learning rate can result in more exploration of the value field potentially leading to better results. This results from the fact that with a smaller learning rate, the stopping criterion will be satisfied slower in comparison with a higher learning rate.

3.2.6. Example

To explain how TD learning works, the same minimization problem will be used as in section 3.1.5. To solve the problem, the 1-step TD method is used. Although it has been explained that the 1-step TD methods needs a decaying learning rate to average over all possible actions in each state, a constant step size $\alpha = 1$ is used to simplify the explanation of the solution to the problem.

As stated before in section 3.2.1 if a greedy policy is used in example I, two sample paths are possible. The first possible sample path is shown in figure 3.19.

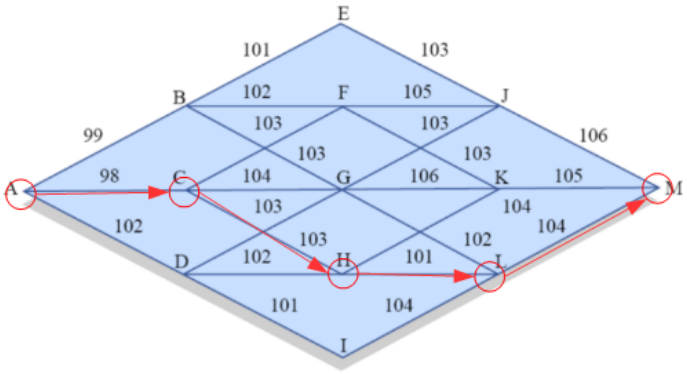


Figure 3.19: Greedy policy sample path scenario I

	States												
Iteration	A	B	C	D	E	F	G	H	I	J	K	L	M
1	98	0	103	0	0	0	0	101	0	0	0	104	0
2	201	0	204	0	0	0	0	205	0	0	0	104	0
3	302	0	308	0	0	0	0	205	0	0	0	104	0
4	406	0	308	0	0	0	0	205	0	0	0	104	0
5	406	0	308	0	0	0	0	205	0	0	0	104	0

Table 3.3: Greedy policy sample path scenario I

When there are multiple actions possible in a state, several scenarios can be used. The algorithm can for example choose the same action every time it visits the specific state. In example I this would mean that after state C every time state F or H is visited. In another scenario a probability factor could be used that determines which action is chosen, meaning that sometimes F is visited after C and sometimes H. Below, the first two scenarios are explained.

It is recalled that the 1-step TD is calculated by:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + V(s_{t+1}) - V(s_t)] \tag{3.49}$$

The same path will be sampled until the stopping criterion is satisfied.

$$\max_{s \in S} |V_{i+1}(s) - V_i(s)| \leq \Delta, \tag{3.50}$$

where $\Delta = 0.001$.

The first time the states are visited the values will be:
$V(A) \leftarrow 0 + [98 + 0 - 0] = 98$
$V(C) \leftarrow 0 + [103 + 0 - 0] = 103$
$V(H) \leftarrow 0 + [101 + 0 - 0] = 101$
$V(L) \leftarrow 0 + [104 + 0 - 0] = 104$
$V(M) = 0$, because it is the terminal state.

When the states are visited for the second time:
$V(A) \leftarrow 98 + [98 + 103 - 98] = 201$
$V(C) \leftarrow 103 + [103 + 101 - 103] = 204$
$V(H) \leftarrow 101 + [101 + 104 - 101] = 205$
$V(L) \leftarrow 104 + [104 + 0 - 104] = 104$
This process is repeated until the stopping criterion is satisfied which happens in iteration 5 as can be seen in table 3.3.

Following a greedy policy can also result in another sample path that is different from the one in figure 3.19. In state C the highest reward can be earned by going to state H but also to state F. By following the path through state F the sample path in figure 3.20 is created. The values for the visited states per iteration are shown in table 3.4.

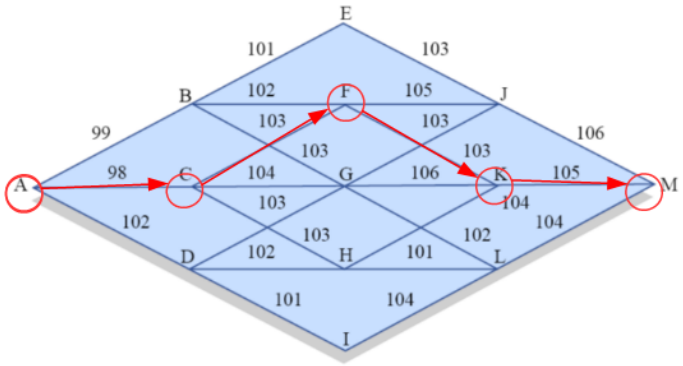


Figure 3.20: Greedy policy sample path scenario II

	States												
Iteration	A	B	C	D	E	F	G	H	I	J	K	L	M
1	98	0	103	0	0	103	0	0	0	0	105	0	0
2	201	0	206	0	0	208	0	0	0	0	105	0	0
3	304	0	311	0	0	208	0	0	0	0	105	0	0
4	409	0	311	0	0	208	0	0	0	0	105	0	0
5	409	0	311	0	0	208	0	0	0	0	105	0	0

Table 3.4: Greedy policy sample path scenario II

Knowing the result from section 3.1.5, following a greedy policy can lead to the optimal result shown in figure 3.19 but can also lead to a solution that is not optimal, as shown in figure 3.20. Without prior knowledge it can not be known if the solution is optimal without exploring the value function. If an ϵ greedy policy is used, more states are visited and if better states exist, depending on exploration probability ϵ, they can be found. In the last two greedy policy examples the learning rate was set equal to 1 to simplify the solution of the problem. By having a different sample path every iteration (second scenario), a learning rate lower than 1 has to be used in order to converge the value function, this can be explained with a simple example.

Assume that $\epsilon = 0.2$, that statistically means that one in every 5 actions is random. For this example it is assumed that the values for all states are known (green indicated numbers) and the stopping criterion is still not met after multiple iterations and therefore more sample paths are needed. Two of these possible sample paths are illustrated in figure 3.21 with I and II. The policy in each state is indicated with black arrows.

When sample path I is followed the calculation for the state value of A will be:

$$\begin{aligned} V(A) &\leftarrow 406 + 1 \cdot [102 + 307 - 406] \\ V(A) &= 409 \end{aligned} \tag{3.51}$$

If the first sample path reaches the goal state (M) and the second sample path is followed the state value of A will be:

$$\begin{aligned} V(A) &\leftarrow 409 + 1 \cdot [98 + 308 - 409] \\ V(A) &= 406 \end{aligned} \tag{3.52}$$

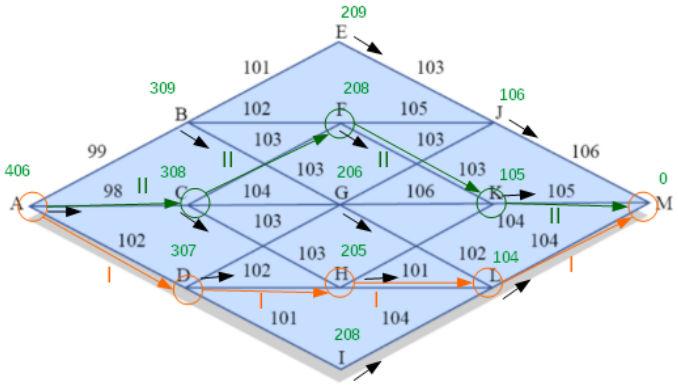


Figure 3.21: TD ϵ-greedy

The values of all the visited states will change continuously depending on the sample paths. In section 3.2.5 the different possible learning rates for the state value function $V(s)$ and action value function $Q(s,a)$ were explained. For the ϵ greedy example, the Q-learning algorithm will be used which calculates the action value function $Q(s,a)$. The same value for ϵ will be used and therefore the two illustrated paths are also possible sample paths for this example. In theory if Q-learning runs infinite iterations, the algorithm will converge to the optimal policy. The values of the optimal solution are shown in table 3.5. It can be seen that the value per state action pair differs and from the table the optimal solution can be deduced by following the action in each state that has the lowest value.

Current State													
Next state	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	0	0	0	0	0	0	0	0	0	0	0	0
B	408	0	0	0	0	0	0	0	0	0	0	0	0
C	406	0	0	0	0	0	0	0	0	0	0	0	0
D	409	0	0	0	0	0	0	0	0	0	0	0	0
E	0	310	0	0	0	0	0	0	0	0	0	0	0
F	0	310	311	0	0	0	0	0	0	0	0	0	0
G	0	309	310	309	0	0	0	0	0	0	0	0	0
H	0	0	308	307	0	0	0	0	0	0	0	0	0
I	0	0	0	309	0	0	0	0	0	0	0	0	0
J	0	0	0	0	209	211	209	0	0	0	0	0	0
K	0	0	0	0	0	208	211	209	0	0	0	0	0
L	0	0	0	0	0	0	206	205	208	0	0	0	0
M	0	0	0	0	0	0	0	0	0	106	105	104	0

Table 3.5: Q-learning values ϵ-greedy

When a stopping criterion is used, this will prevent the algorithm of doing an infinite amount of iterations. It therefore will not have the exact values of the optimal solutions but an approximation as can be seen in table 3.6. The stopping criterion used to determine the values shown in table 3.6 is a small value as explained before in section 3.2.1. In this simulation it was set equal to $\Delta = 0.001$. Although the values in the simulation are an approximation, the policy found is equal to the optimal policy as can be deduced from the table.

Current State													
Next state	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	0	0	0	0	0	0	0	0	0	0	0	0
B	406.524	0	0	0	0	0	0	0	0	0	0	0	0
C	405.994	0	0	0	0	0	0	0	0	0	0	0	0
D	407.730	0	0	0	0	0	0	0	0	0	0	0	0
E	0	308.894	0	0	0	0	0	0	0	0	0	0	0
F	0	308.501	310.340	0	0	0	0	0	0	0	0	0	0
G	0	308.204	309.390	307.104	0	0	0	0	0	0	0	0	0
H	0	0	307.994	306.581	0	0	0	0	0	0	0	0	0
I	0	0	0	306.581	0	0	0	0	0	0	0	0	0
J	0	0	0	0	208.883	210.200	208.740	0	0	0	0	0	0
K	0	0	0	0	0	207.975	209.104	208.890	0	0	0	0	0
L	0	0	0	0	0	0	205.994	204.995	207.720	0	0	0	0
M	0	0	0	0	0	0	0	0	0	106	105	104	0

Table 3.6: Q-learning values ϵ-greedy

Eligibility Traces

In section 3.2.3 it has been explained that eligibility traces propagate a 'share' of the rewards gained during the episode. In other words, a certain reward has been earned by visiting the states in an episode and therefore these states are considered 'good' because they lead to a reward. Assume the same greedy policy is followed resulting in the sample paths seen in figure 3.22 using accumulating traces.

It is recalled that:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$

For this example, the value for the trace is chosen to be λ=0.7, γ=1 and learning rate α=1. This value for the learning rate has been chosen to be able to compare the state values with the TD method. Per episode 5 states are visited by the 'agent' of the algorithm. When the agent goes from one state to the next it leaves a trace, the resulting eligibility traces can be seen in table 3.7. In the first column of the table, the stages (m=1-4) of the agent are indicated. In the first step of the first episode the agent goes from state A to state D. This is also seen in table 3.7 in which the trace of D becomes 1 in stage 2, while the trace of A decays to $0.700 \cdot 1 = 0.700$. If the agent moves from D to H, it can be seen that the trace of H becomes 1 while traces of A and D decay to 0.490 respectively 0.700. This process is repeated for all three episodes in this example resulting in table 3.7.

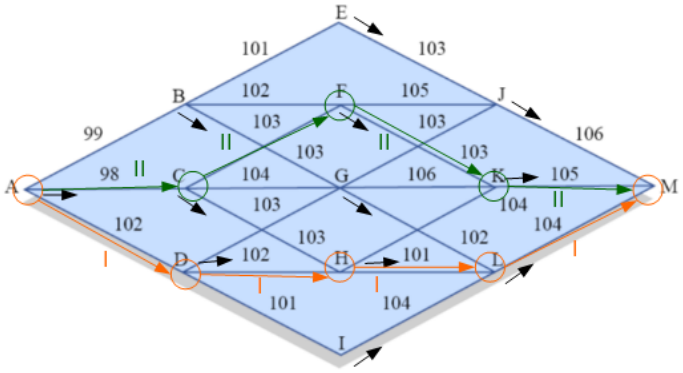


Figure 3.22: TD ϵ-greedy eligibility traces

States														
Stage	A	B	C	D	E	F	G	H	I	J	K	L	M	Episode
1	1.000	0	0	0	0	0	0	0	0	0	0	0	0	I
2	0.700	0	0	1.000	0	0	0	0	0	0	0	0	0	
3	0.490	0	0	0.700	0	0	0	1.000	0	0	0	0	0	
4	0.343	0	0	0.490	0	0	0	0.7000	0	0	0	1.000	0	
1	1.343	0	0	0.343	0	0	0	0.490	0	0	0	0.700	0	
2	0.940	0	1.000	0.240	0	0	0	0.343	0	0	0	0.490	0	II
3	0.658	0	0.700	0.168	0	1.000	0	0.240	0	0	0	0.343	0	
4	0.461	0	0.490	0.118	0	0.700	0	0.168	0	0	1	0.240	0	

Table 3.7: Eligibility traces

If the values of the states are calculated on-line, the last steps of algorithm 6 are applied:
For all s:
$V(s) \leftarrow V(s) + \alpha\delta e(s)$
$e(s) \leftarrow \gamma\lambda e(s)$

Every state the agent visits, it determines $\delta_t = R_t^{(k)} - \bar{V}_t(s_t)$. This temporal difference error δ_t (for $k = 1$) is used with the eligibility trace to determine the values of the states, which results in the values in table 3.8. It can be seen that the values of each state change substantially after each step of the agent. Depending on learning rate α, λ and the k-steps return, the values of the states will be different, however the solution (the path through the value field) will be the same. The algorithm will stop when the earlier mentioned stopping criterion is satisfied.

States														
Stage	A	B	C	D	E	F	G	H	I	J	K	L	M	Sample Path
1	102	0	0	0	0	0	0	0	0	0	0	0	0	I
2	173.400	0	0	102	0	0	0	0	0	0	0	0	0	
3	222.890	0	0	172.700	0	0	0	101	0	0	0	0	0	
4	258.562	0	0	223.660	0	0	0	173.800	0	0	0	104	0	
1	42.937	0	0	168.587	0	0	0	95.125	0	0	0	-8.393	0	
2	139.757	0	103	193.317	0	0	0	130.454	0	0	0	42.076	0	II
3	207.539	0	175.100	210.629	0	103	0	155.184	0	0	0	77.406	0	
4	255.907	0	226.550	222.982	0	176.500	0	172.831	0	0	105	102.616	0	

Table 3.8: Value function with eligibility traces

In this section the off-policy (Q-Learning) and on-policy (Sarsa) Temporal Difference learning algorithms and the variants with eligibility traces(Q-λ, Sarsa λ) have been discussed as potential techniques for the optimal route synthesis. In section 3.3 the different algorithm are used to create trajectories in several scenarios. These trajectories will be compared in order to find the best algorithm to create the target trajectories for the emergency trajectory planner.

3.3. Selecting the route planning algorithm

In order to find out which algorithm creates the most optimal routes in the shortest time period, they are compared with trajectories generated by an A* (star) algorithm. The A* algorithm (Hart et al., 1968) is one of the most widely used algorithms for shortest-path problems and will always find the optimal trajectory if certain conditions are met (Korf, 1990). In this chapter the A* algorithm will be explained and compared with the TD algorithms.

3.3.1. A* Algorithm

Beginning at the starting state, the surrounding states are checked if these are accessible or if there are obstacles in these surrounding states. The states that are accessible are added to a list and marked 'open'; this list can be seen as a list of states that have to be checked. For every state on the 'open' list the following cost function has to be calculated:

$$f(s) = g(s) + h(s), \tag{3.53}$$

where $g(s)$ is the actual cost of an optimal path from the start state to the current state s and $h(s)$ is the actual cost of an optimal path from current state s to the goal state.

The real values for f, g and h until this point are unknown and therefore have to be estimated by:

$$\hat{f}(s) = \hat{g}(s) + \hat{h}(s), \tag{3.54}$$

were $\hat{f}, \hat{g}, \hat{h}$ are the estimated values of f, g and h.

The next state is the adjacent state with the lowest value for f. When a state is visited, it is put on the 'closed' list. The adjacent states of the current state are added to the 'open' list (if they are not already there). For all adjacent states on the 'open' list the value for f is determined. The next state is again chosen based on the lowest value for f. This process is repeated until the A* algorithm puts the goal state on the 'closed' list or if it fails to find the goal state, which means no path is found. The first condition for which the A* algorithm finds the optimal path is, that a path from start to goal exists. The second condition is $\hat{h} \leq h$, this means that $\hat{h}$, the estimate of h, has to underestimate the value of h in order to find the optimal path.

3.3.2. Experiment

The objective of this experiment is to determine which ADP algorithm performs the best in generating target trajectories and will therefore be used in the emergency trajectory planner. It is recalled that these trajectories have to be generated in a short amount of time to be able to use it on line.

Due to the fact that the ADP algorithms with eligibility traces have to update the trace for all states after each step, the algorithm is multiple times slower making it unsuitable for the purpose of generating trajectories in emergency situations. Q-λ and Sarsa-λ will therefore not be examined in the experiment.

As stated in section 3.2 both Q-learning (Sutton and Barto, 1998) and Sarsa (van Seijen et al., 2009) converge to the optimal policy and value function $Q(s,a)$ if all states are updated an unlimited amount of times. A practical solution for Q-learning to converge is to use a stopping condition $\max_{s \in S} |Q_{i+1} - Q_i| \leq \Delta$ where Δ has a small value. When a stopping condition is used, it is not guaranteed that Sarsa or Q-learning will find the optimal policy. Important factors to increase the chance of finding the optimal policy are the learning rate α and the exploration factor ϵ. If a low learning rate is used it will take longer until the value function is converged, which in turn enables more steps to be taken to explore the environment. If an ϵ-greedy policy is used, the amount of random steps is determined by ϵ, which is the probability of taking a random step. If only greedy steps are used, the value field will converge only to local optima, resulting in a non optimal solution.

To determine the effects of learning rate α and exploration probability ϵ on the performance of the algorithms, multiple simulations will be done. To give a statistically significant measure of performance, the number of simulations is 100 (Caironi and Dorigo, 1994).

The above mentioned algorithms and the A* will all generate a path in two different scenarios from the same

predefined start and goal in a 100x100 randomly generated grid in which 130 random obstacles are created. In figure 3.23 can be seen that some obstacles, portrayed as green squares, are overlapping because as stated before they are generated randomly.

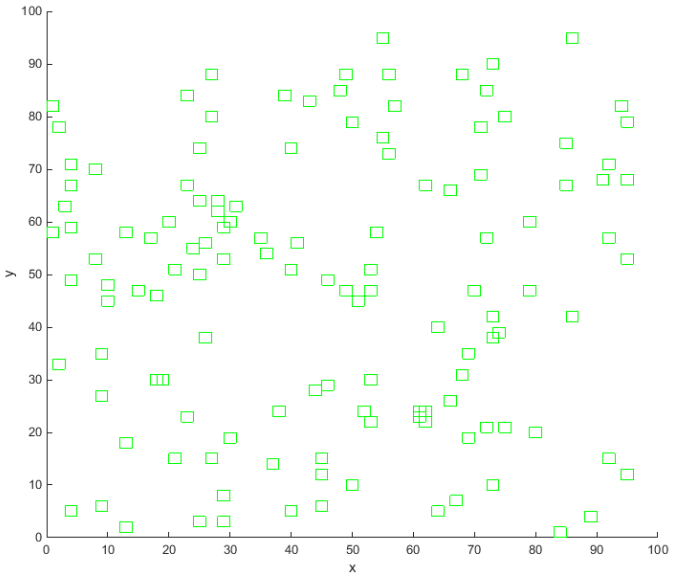


Figure 3.23: Random generated comparison grid

The eight possible actions for the ADP algorithms and the A* algorithm are the same and are displayed in figure 3.24.

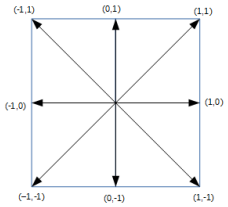


Figure 3.24: Possible actions

The reward function of the ADP algorithms is defined by rewards, penalties and the euclidean distance to the goal:

$$\sqrt{(x+\Delta x_{action}-x_{goal})^2+(y+\Delta y_{action}-y_{goal})^2}, \tag{3.55}$$

where x and y are the coordinates of the current state.

The action that leads to a state with an obstacle receives a penalty (positive value) and a reward for reaching the goal (negative value). It is recalled that all problems in this research are minimization problems meaning that a path through states with the lowest values (all rewards that can be earned from that state to the goal) is optimal. If there are no obstacles, the action that brings the agent the closest to the goal will get the lowest reward. When a state has an obstacle, the value of the action that leads to that state will be much higher than action that lead to states without obstacles. The same holds for actions that lead to the goal state. The penalties and rewards also influence the paths found by the ADP algorithms. If the penalty is low in comparison with the euclidean distance, the algorithm will not try to go around the obstacle but go straight through it. If the penalty is too high, the algorithms will not be able to create paths close to an obstacle. In order words, the value of the penalties influences the paths found by the ADP algorithm but the obstacles will remain soft obstacles. In contrary to the ADP algorithms, the A* algorithm used in this research assumes the obstacles are hard constraints meaning no paths through them are possible.

Earlier it was explained how the A* algorithm works and it was stated that the A* paths are the optimal solution.

The policy found by the ADP algorithms determine at which state, which next state is chosen resulting in a path. To compare the routes created by ADP algorithms with the A*, the total rewards hypothetically earned by following the A* path are summed and compared with the summed rewards for the policies found by Sarsa and Q-learning. For each value of ϵ [0.01,0.1,0.2,0.5] 4 different learning rates [0.2,0.4,0.6,0.8] are used to show the influence of the learning rate on convergence with a constant exploration probability. The data generated in this experiment will be analyzed by using *boxplots* in which the median is indicated with the red horizontal lines and the outliers with red crosses. The reward of reaching the goal is the same for both scenarios and is equal to -9000. The reward is chosen to be a high value to speed up convergence. The penalty for visiting an obstacle state in both scenarios is 60.

3.3.3. Scenario I

In the first scenario a short path has to be created. The path created by the A* algorithm is shown in figure 3.25. The total accumulated reward for the A* path in the first scenario is -8949 with a simulation time of 1.2 seconds.

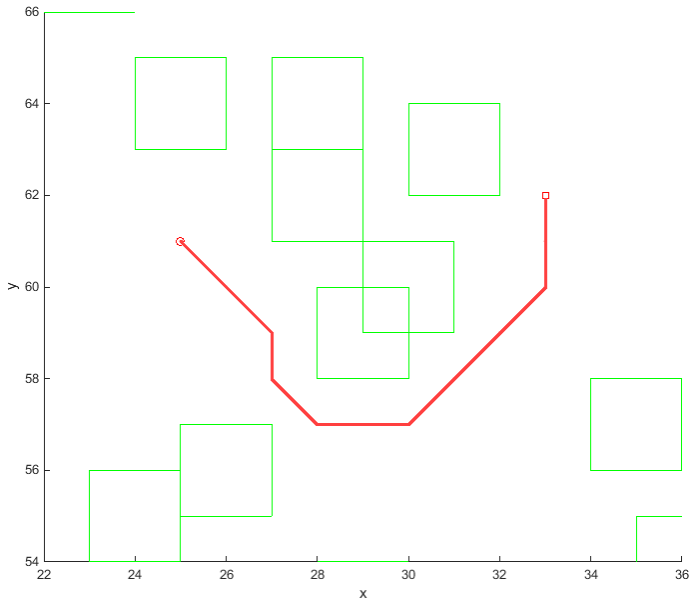
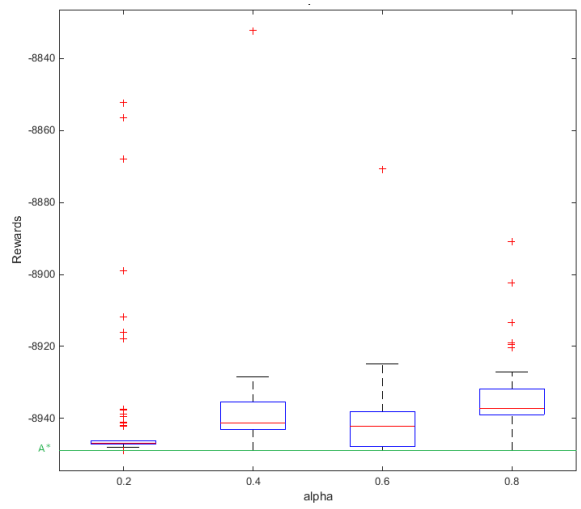


Figure 3.25: A* path scenario I

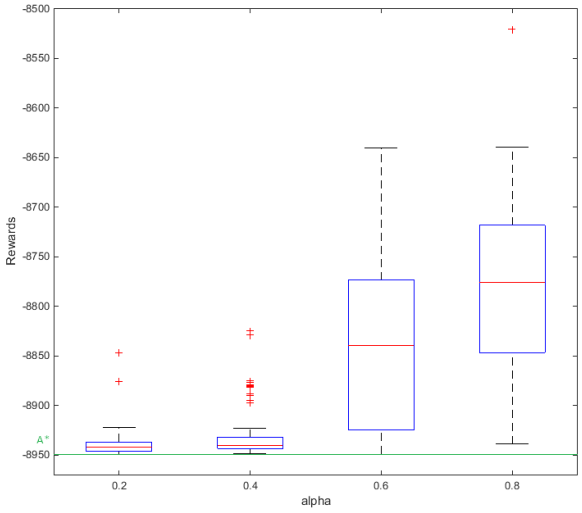
The results of the simulations for Sarsa are shown in figure 3.26 and for Q-learning in figure 3.27. The simulation time of Sarsa (table 3.9a) and Q-learning (table 3.9b) are also displayed, to be able to identify the best algorithm based on accuracy and simulation time.

The objective of this experiment is to find an algorithm that produces results close to the optimal solution provided by the A* algorithm indicated in figures 3.26 and 3.27. In the two figures it can be seen that increasing ϵ, the probability of an exploration step, leads to more spread in the results for Sarsa as well as Q-learning. The results for both algorithms for $\epsilon = 0.5$ are substantially worse in comparison with the other values for ϵ and the spread is higher. It will therefore not be analyzed further. Note that the sub-figures in figures 3.26 and 3.27 have different scales, therefore in figure 3.28 the results for Sarsa and Q-learning for $\epsilon = 0.01$, 0.1, 0.2 will be displayed on the same scale to better compare the results.

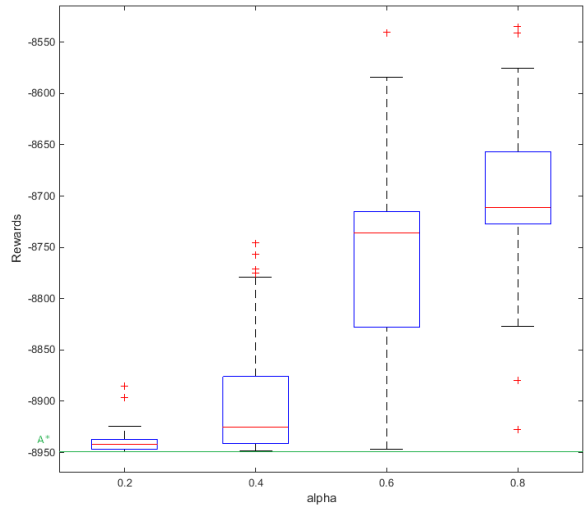
From figure 3.28 it can be deduced that the performance of Sarsa and Q-learning for $\epsilon = 0.01$ are comparable, however Q-learning uses a shorter calculation time period to produce a slightly better result. The difference between Sarsa and Q-learning starts to show for $\epsilon = 0.1$ and $\epsilon = 0.2$. In figure 3.28c and 3.28d the results for $\alpha = 0.2$ and $\alpha = 0.4$ are comparable however for $\alpha = 0.6$ and $\alpha = 0.8$ the performance of Sarsa decreases substantially.



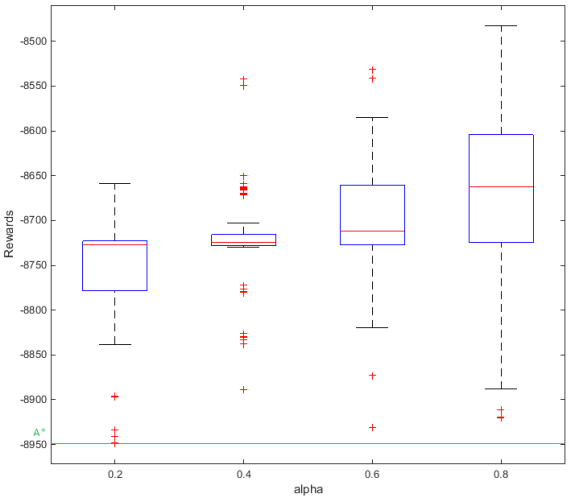
(a) $\epsilon = 0.01$



(b) $\epsilon = 0.1$

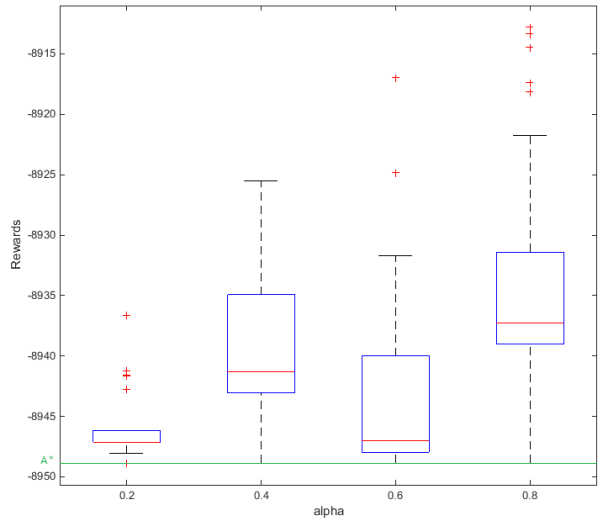


(c) $\epsilon = 0.2$

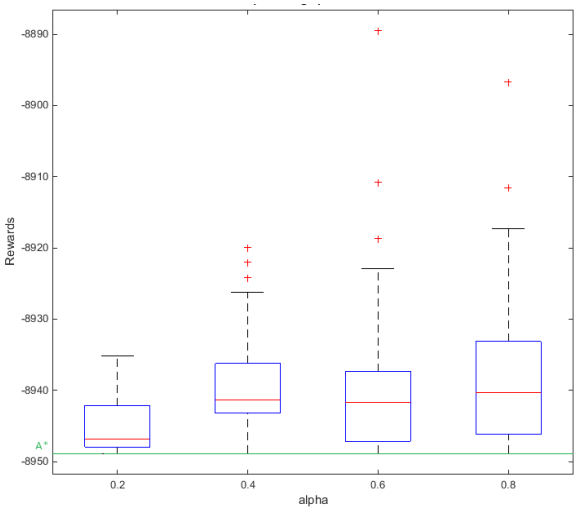


(d) $\epsilon = 0.5$

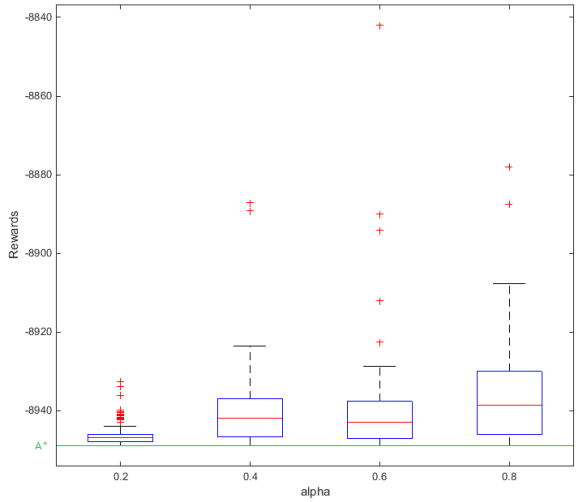
Figure 3.26: Comparing ϵ Sarsa scenario I



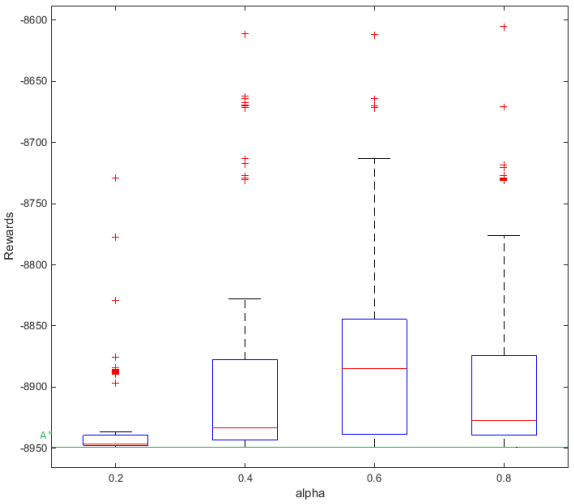
(a) $\epsilon = 0.01$



(b) $\epsilon = 0.1$

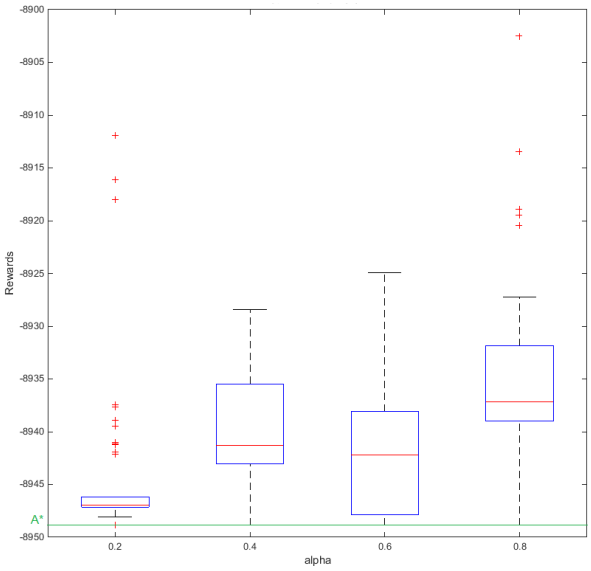


(c) $\epsilon = 0.2$

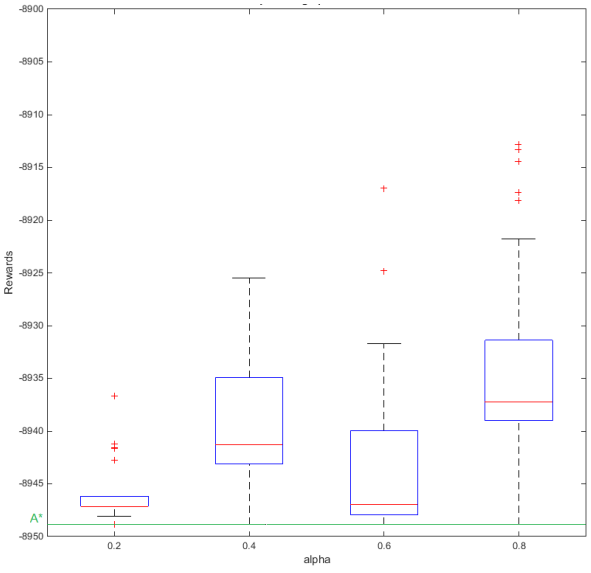


(d) $\epsilon = 0.5$

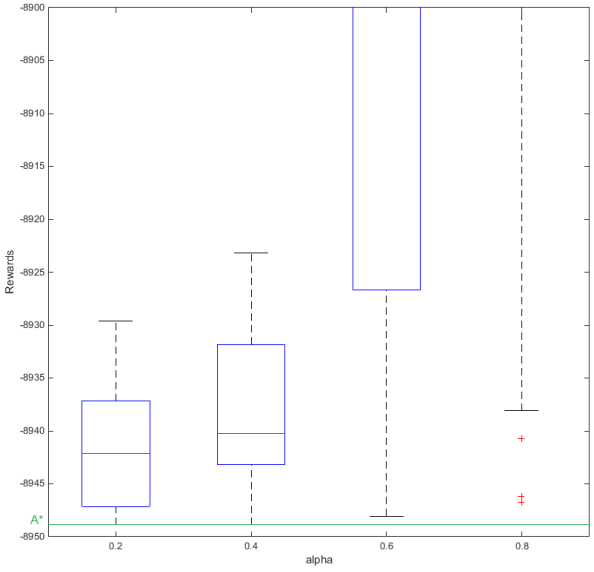
Figure 3.27: Comparing ϵ Q-learning scenario I



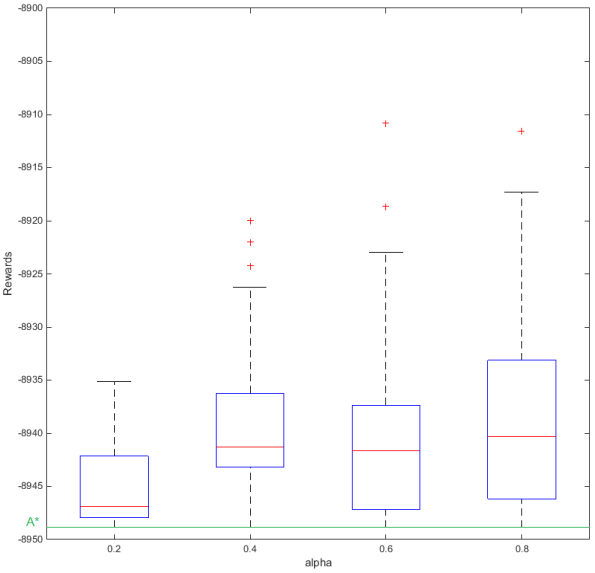
(a) Sarsa $\epsilon = 0.01$



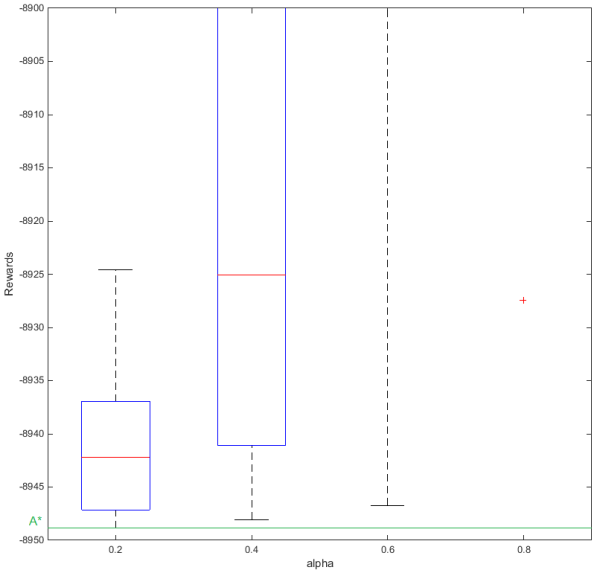
(b) Q-learning $\epsilon = 0.01$



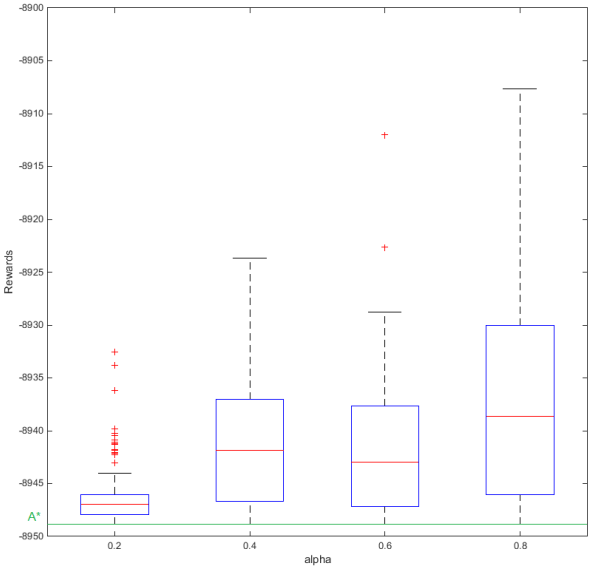
(c) Sarsa $\epsilon = 0.1$



(d) Q-learning $\epsilon = 0.1$



(e) Sarsa $\epsilon = 0.2$



(f) Q-learning $\epsilon = 0.2$

Figure 3.28: Comparing Sarsa and Q-learning scenario I

ϵ	α	time(s) / (100 simulations)
0.01	0.2	13
0.01	0.4	13
0.01	0.6	14
0.01	0.8	15
0.1	0.2	28
0.1	0.4	42
0.1	0.6	87
0.1	0.8	169
0.2	0.2	62
0.2	0.4	109
0.2	0.6	218
0.2	0.8	374
0.5	0.2	732
0.5	0.4	1472
0.5	0.6	3091
0.5	0.8	5924

(a) Sarsa

ϵ	α	time(s) / (100 simulations)
0.01	0.2	9
0.01	0.4	10
0.01	0.6	11
0.01	0.8	11
0.1	0.2	10
0.1	0.4	10
0.1	0.6	10
0.1	0.8	10
0.2	0.2	11
0.2	0.4	10
0.2	0.6	11
0.2	0.8	11
0.5	0.2	12
0.5	0.4	13
0.5	0.6	13
0.5	0.8	14

(b) Q-learning

Table 3.9: Scenario I simulation time

In figure 3.28f it can be seen that the results of Q-learning for $\epsilon = 0.2$ are comparable to the other exploration rates for Q-learning but the spread of the boxplot is slightly higher. The results for Sarsa keep on deteriorating in comparison with the other exploration rates and all results of Q-learning. Comparing the algorithm with respect to simulation time, it can be deduced from tables 3.9a and 3.9b that for $\epsilon = 0.01$, it is slightly higher for Sarsa (13-15 seconds) compared to Q-learning (9-11 seconds). For $\epsilon = 0.1$, 0.2 the simulation time of Sarsa is multiple times higher than for Q-learning.

3.3.4. Scenario II

To compare the results between Sarsa and Q-learning for a longer path, a second scenario is used. For the second scenario the results are not tested for $\epsilon = 0.5$ because in scenario I it could be clearly seen that having a higher exploration probability produces worse results for both Sarsa and Q-learning in comparison with the lower exploration probabilities ($\epsilon = 0.01, 0.1, 0.2$). Also as stated earlier the results for $\epsilon = 0.2$ in comparison with $\epsilon = 0.01, 0.1$ had a slightly higher spread and longer simulation time and is therefore not tested in scenario II. Learning rate $\alpha = 0.8$ is also not tested in the second scenario because it could be concluded from scenario I that the performance is substantially worse than for learning rate 0.2, 0.4 and 0.6 in respect to accumulated rewards and simulation time.

The trajectories made by Sarsa and Q-learning will again be compared with the A* algorithm solution shown in figure 3.29. The total accumulated rewards for the A* path in the second scenario is -8794 with a simulation time of 1.4 seconds.

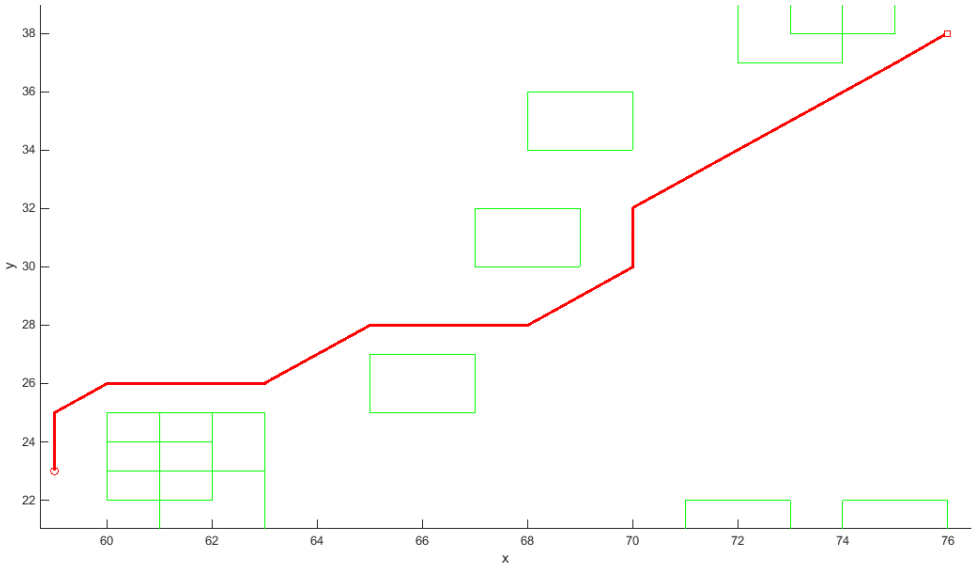


Figure 3.29: A* path scenario II

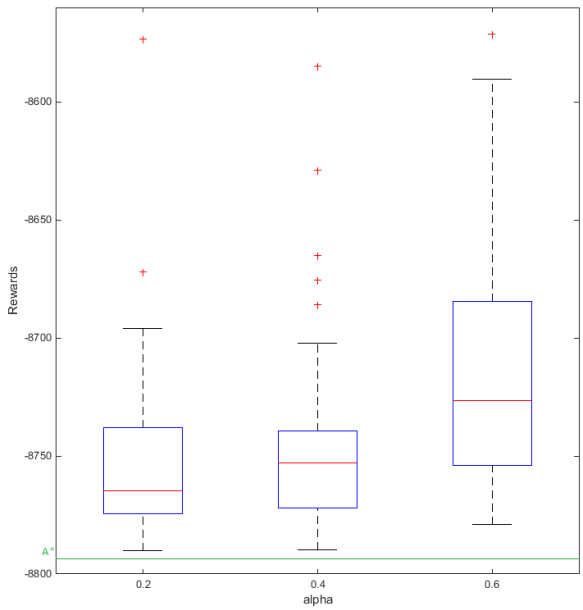
ϵ	α	time (s) / (100 simulations)
0.01	0.2	211
0.01	0.4	161
0.01	0.6	232
0.1	0.2	593
0.1	0.4	4563
0.1	0.6	20504

(a) Sarsa

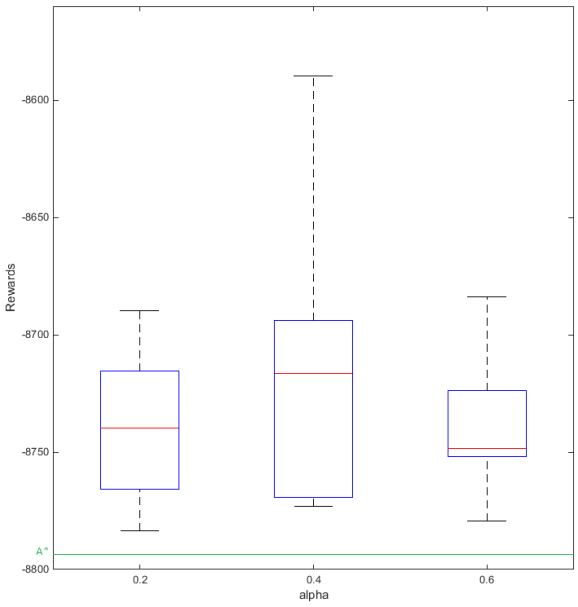
ϵ	α	time (s) / (100 simulations)
0.01	0.2	125
0.01	0.4	98
0.01	0.6	89
0.1	0.2	126
0.1	0.4	100
0.1	0.6	95

(b) Q-learning

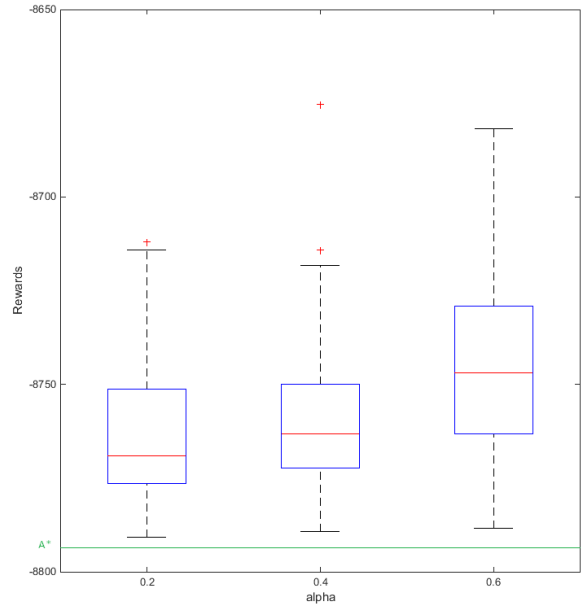
Table 3.10: Scenario II simulation time



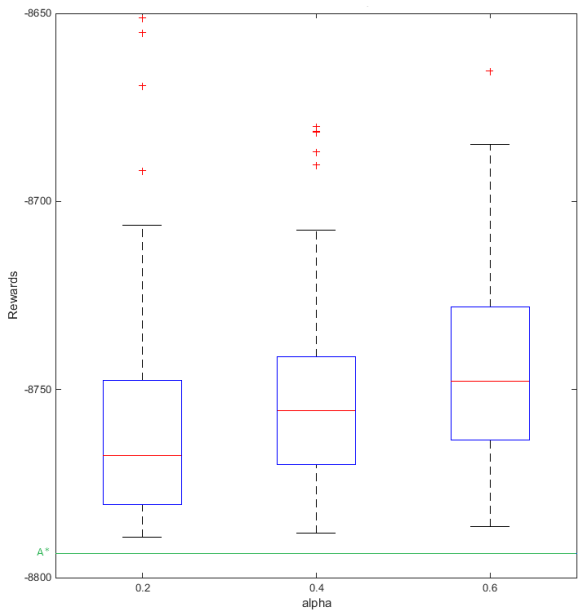
(a) Sarsa $\epsilon = 0.01$



(b) $\epsilon = 0.1$

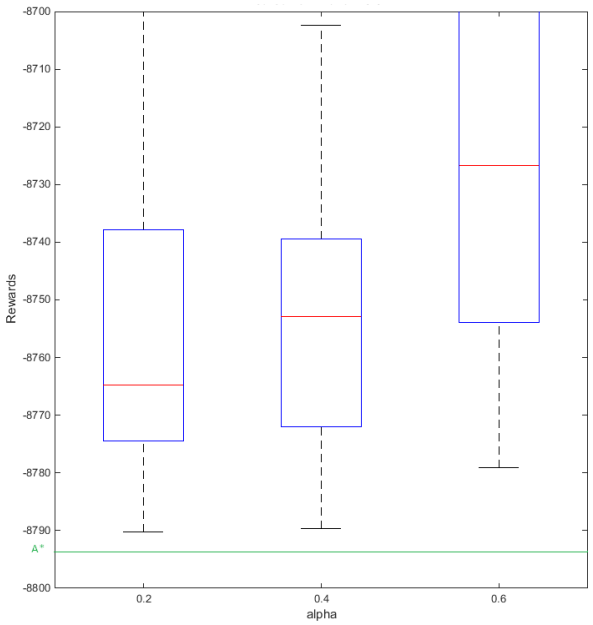


(c) $\epsilon = 0.01$

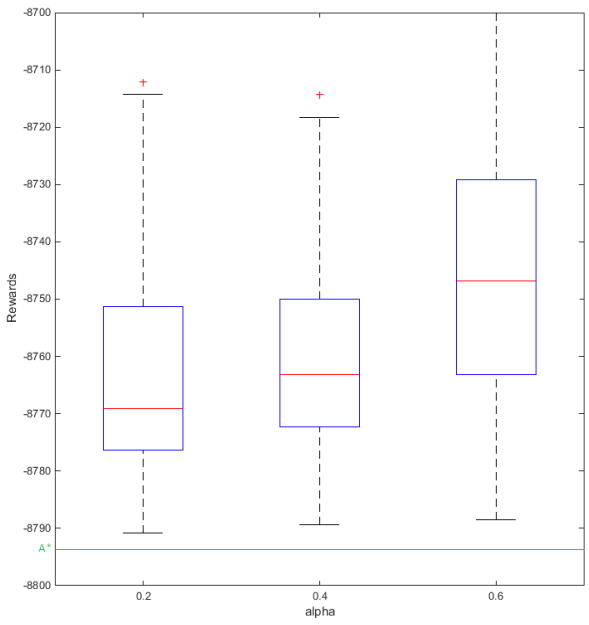


(d) $\epsilon = 0.1$

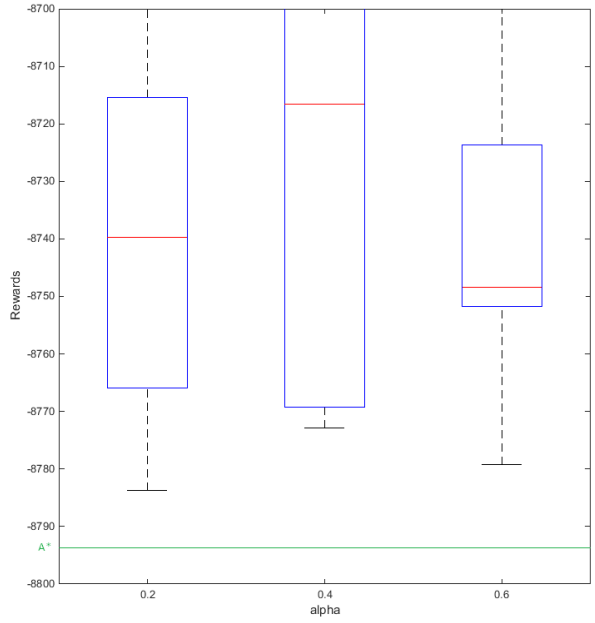
Figure 3.30: Sarsa and Q-learning scenario II



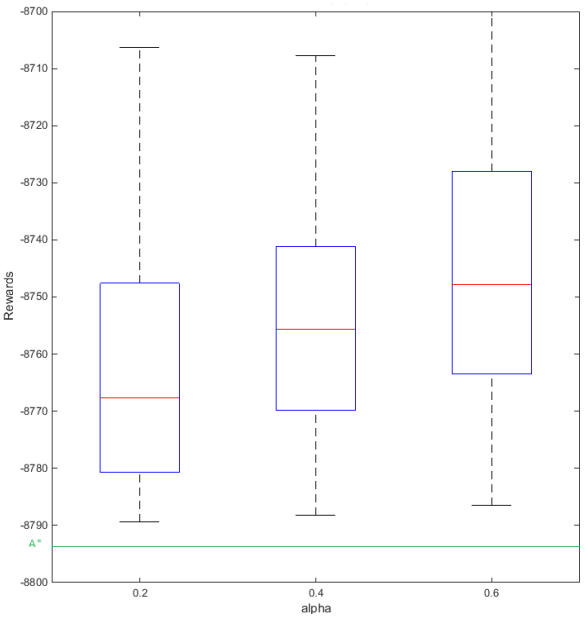
(a) Sarsa $\epsilon = 0.01$



(b) Q-learning $\epsilon = 0.01$



(c) Sarsa $\epsilon = 0.1$



(d) Q-learning $\epsilon = 0.1$

Figure 3.31: Sarsa and Q-learning scenario II adjusted scale

The results of the simulation for Sarsa and Q-learning are shown in figure 3.30. Note that the subfigures have different scales. To better compare the results of the two algorithms the values are also shown on the same scale in figure 3.31. From it can be deduced that like in scenario I, Sarsa performs worse than Q-learning. Not only the spread of Sarsa is higher for all learning rates and exploration probabilities but also the simulation time is multiple times higher.
When analyzing the results of Q-learning, it can be seen that the simulation time, shown in table 3.10b, for $\epsilon = 0.01$ is lower for all learning rates than for $\epsilon = 0.1$. This can be explained by the fact that $\epsilon = 0.01$ means less exploration steps which in turn results in faster convergence.

Analyzing the differences between the results of $\alpha = 0.2$ and $\alpha = 0.4$ for $\epsilon = 0.01$ and $\epsilon = 0.1$ for Q-learning taking into account the generation time, the best results for both $\epsilon = 0.01$ and $\epsilon = 0.1$ are achieved for learning rate $\alpha = 0.4$. Although the results for $\epsilon = 0.01$ are better for environments with small obstacles, an exploration probability has to be chosen that can cope with the large obstacles defined in section 2.2.1. Therefore $\epsilon = 0.1$ is chosen to create the target trajectories in the trajectory planner defined in chapter 2.

When the performance of Q-learning with $\epsilon = 0.1$ and $\alpha = 0.4$ is compared with A* it can be seen in figure 3.31d that the paths created by Q-learning are spread while A* finds the optimal path. It is however recalled from chapter 2 that the trajectory generated by the Q-learning algorithm is a target trajectory which will be used by the dynamic model to make a 3D path. It therefore does not have to be exactly equal to the optimal solution found by the A* algorithm but can have a spread like shown in figure 3.31d. Comparing the simulation times in table 3.10b it can be seen that they are comparable, with 1.4 second for the A* and an average of 1 second for $\epsilon = 0.1$ and $\alpha = 0.4$.

From the results of the experiment it was concluded that the Q-learning algorithm with exploration probability $\epsilon = 0.1$ and learning rate $\alpha = 0.4$ gives the best results of the tested algorithms when creating trajectories. In the next chapter it will be explained how the Q-learning algorithm is used to create trajectories in the trajectory planner introduced in chapter 2.

4

Model Specifications

4.1. Target Trajectory Generation

In section 3.2 several route synthesis algorithms have been explained. From the experiment in section 3.3 it was concluded that of the algorithms that were tested, Q-learning created the most optimal trajectories and in the shortest amount of time. Therefore, Q-learning is chosen to create the horizontal target trajectories for the model. The pseudocode from section 3.2 is repeated in algorithm 8.

Algorithm 8: Q-learning (Sutton and Barto, 1998)

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat for (each step of episode):
    Choose a from s using policy derived from Q (e.g., $\epsilon - greedy$)
    Take action a, observe reward, r and next state, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \min_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  until s is terminal

In each state for each action, a separate value $Q(s,a)$ is calculated. Q-learning starts by initializing the value field in which the values $Q(s,a)$ are determined in each state for every action. The initial value field is set equal to 0 for all $Q(s,a)$. Starting in state s, the 'agent' of the algorithm receives a reward by taking a certain action a and moving to a specific next state. The reward function in this model is created by using the euclidean distance between the current state to the goal state:

$$\sqrt{(x + \Delta x_{action} - x_{goal})^2 + (y + \Delta y_{action} - y_{goal})^2}, \tag{4.1}$$

where x and y are the coordinates of the current state.
The potential next states are defined by the possible actions. The model employed here in has 8 possible actions, portrayed in figure 4.1.

Penalties are given to actions that lead to obstacles and rewards for actions that lead to the goal state. The obstacles in the model are the 100 clusters, explained in section 2.2.1 in which it is assumed that the whole population of The Netherlands reside. The magnitude of the value for all obstacles states influences the generated trajectories. To make these states 'unattractive' for the Q-learning agent, the value for all the obstacle states is a high positive value. The obstacles are considered soft constraint because, although the magnitude of the penalty makes the obstacles more or less 'unattractive', the obstacles do not become impenetrable. The value of the reward is a large negative number.

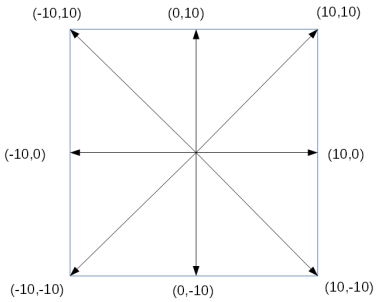


Figure 4.1: Possible actions ADP model

In the system architecture in figure 2.1 in chapter 2 it can be seen that the trajectories generated with Q-learning will be the target trajectories which will be used by the dynamic model to create smoothed routes. In section 2.2.1 it was stated that the resolution of the map is 0.01 km^2 meaning that a step between for example state $(1,1)$ and $(1,2)$ in reality is 100 meters. The trajectories have to be created fast in emergency situations. For aircraft maneuvers there is no need to create a route with a precision of 100 meters and therefore the step-size used is 10 which is equal to a kilometer in reality.

Trajectories are generated to all the runways in the footprint. First a direct route is generated to the runway. If the trajectory satisfies the flight path constraint $[\gamma_{min}, \gamma_{max}]$ a 3D trajectory is made by the dynamic model. If the required flight path angle is too steep, the trajectory has to be created in two parts. The first segment is generated between the aircraft position and an energy dissipation sector. In these sectors, explained in section 2.2.3, the aircraft can dissipate excess speed and altitude. The second segment of the trajectory is created from the energy dissipation sector to the chosen reachable runway. An example of a route is shown in figure 4.2. It can be seen that the target trajectory is not generated to the runway but to the target point which is explained in section 2.2.2.

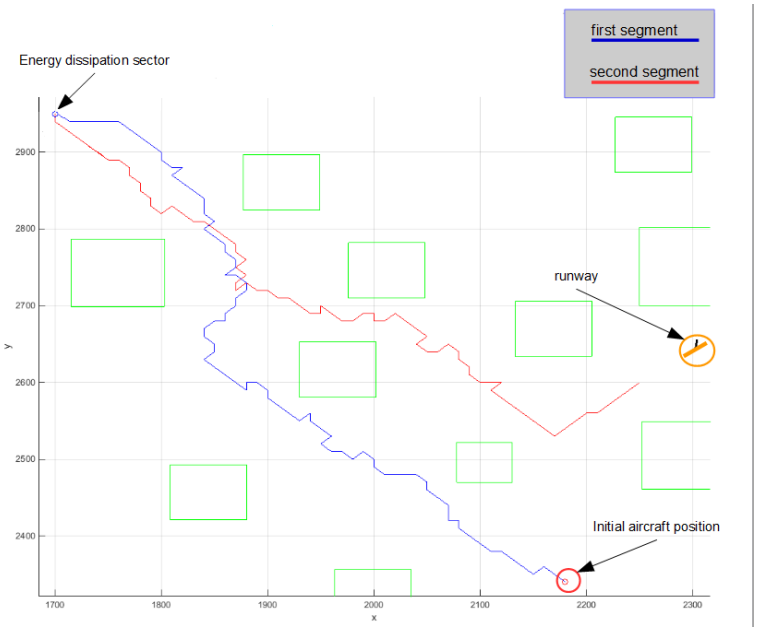


Figure 4.2: Example 2 segments trajectory

This process of creating trajectories is done for every reachable airport, which are all the airports in the footprint circle, defined in section 2.1.

4.2. Penalty Value

The value of the penalty given to actions that lead to obstacles, influences the generated trajectories. In figure 4.3 three different trajectories generated by the Q-learning algorithm and one by the A* algorithm are shown. The three Q-learning trajectories are generated by using different values for the penalty. The trajectory created with penalty=0 does not circumvent the obstacles because there is no penalty for visiting a obstacle state. When the penalty has a value of 60, the trajectory generated avoids the singular obstacles but creates paths through the cluster of obstacles. If a penalty of 6000 is used a trajectory is generated which, as can be seen in figure 4.3, avoids the cluster of obstacles completely following the A* path.

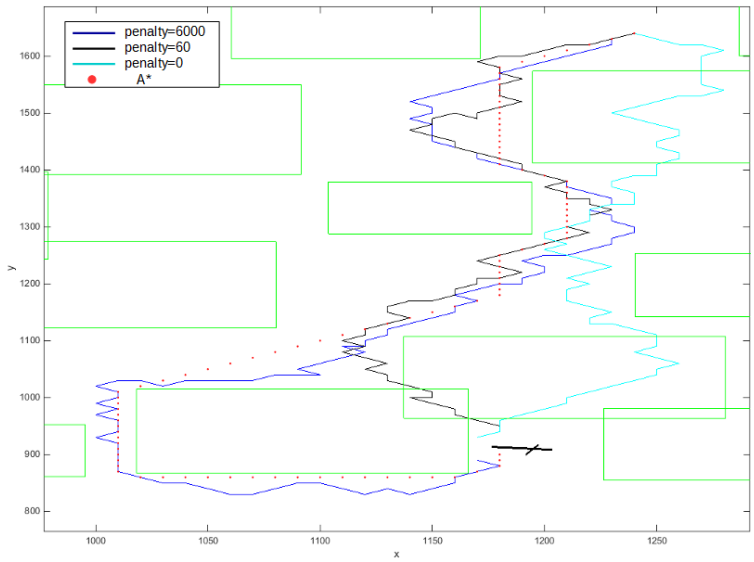


Figure 4.3: Influence of penalties on trajectories

As stated in section 2.2.2, the trajectories generated must be a compromise between population avoidance and shortest path. In figure 4.3 can be seen that a value for the penalty that satisfies this condition is 60, therefore this value is used for the penalties of the model.

5

Results

In this chapter the trajectories generated by the emergency trajectory planner are shown for a scenario in which an aircraft looses all thrust when flying over the *IJsselmeer*. The random chosen initial aircraft position and the corresponding footprint are indicated in figure 5.1. The Q-learning trajectories are benchmarked using the A* algorithm, which as noted in section 3.3 generates the shortest paths and uses obstacles as hard constraints.

As stated in chapter 3, the target trajectories made by Q-learning depend on multiple factors. Even with predetermined values for the exploration probability ϵ and learning rate α the routes will be different every time. For several reachable airports, the Q-learning algorithm will create 5 trajectories between the same start and goal state. For each Q-learning trajectory and A* trajectory the dynamic model will make the resulting route. The length of the ground track of the resulting routes created with the Q-learning target trajectories will be compared with the length of the ground track of the benchmark created with the A* target trajectories.

For the 3D trajectory of the 5 trajectories created, only the most optimal is shown. The trajectories are generated until the *target point* indicated in figure 2.6 at which the aircraft should be stabilized on the glide slope. This trajectories are shown for several airports. It must, however, be noted that the A* algorithm cannot generate trajectories to all the runways due to the fact that some target points are located in obstacles. On these routes the Q-learning trajectories cannot be compared and are therefore not generated. Even though the generated trajectories are not shown for all airports and runways, all runways will be ranked. The plots of all the trajectories generated can be found in appendix A.1. Firstly the generated trajectories to the reachable airports in the horizontal plane are discussed followed by the trajectories in the vertical plane. The chapter ends with the ranking of the trajectories.

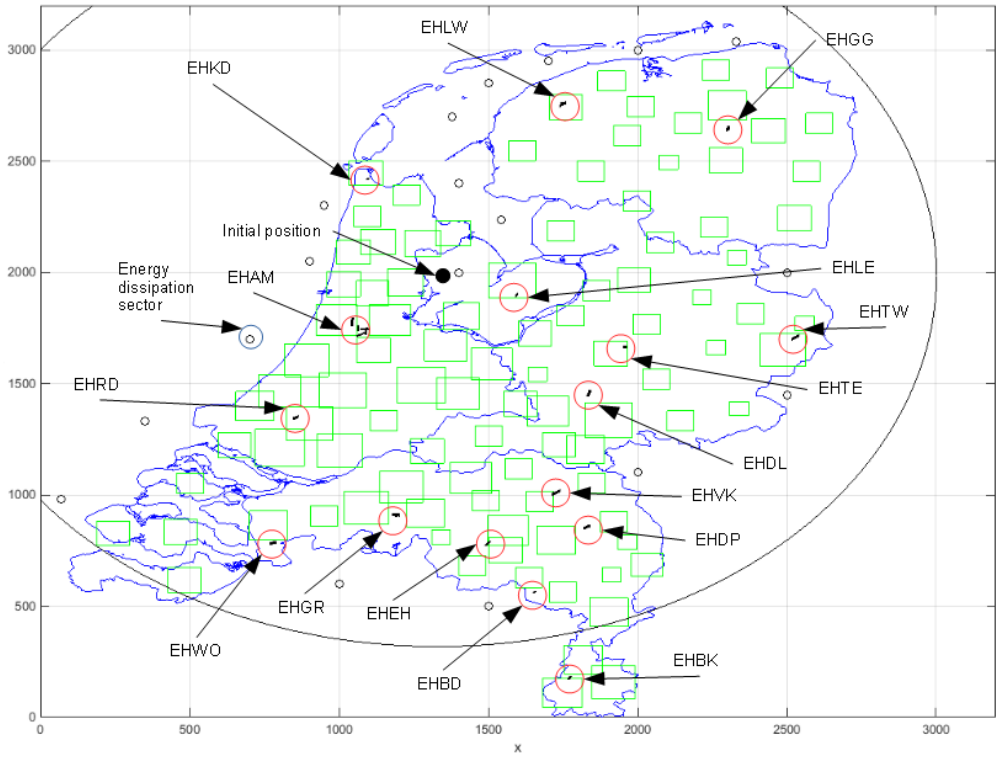
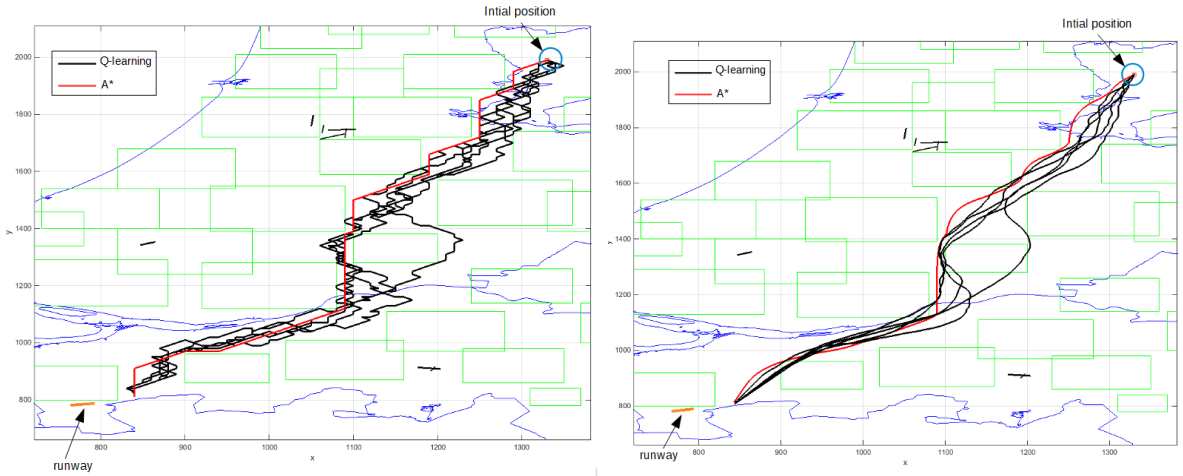


Figure 5.1: Map with footprint

5.1. Horizontal Trajectories

In figure 5.2a and figure 5.2b, respectively, the target trajectories and the smoothed resulting trajectories to runway 25 at Woensdrecht airbase are displayed. In figure 5.2a, the five trajectories generated by the Q-learning algorithm are shown. As can be seen, all trajectories are different and for this specific route have path lengths varying between 103% and 108% of the A* trajectory. For each route from the initial position to the reachable airport, indicated with the orange line, these 5 trajectories are generated. In the next figures however, only the most optimal Q-learning trajectories are shown and compared with the A* trajectories.



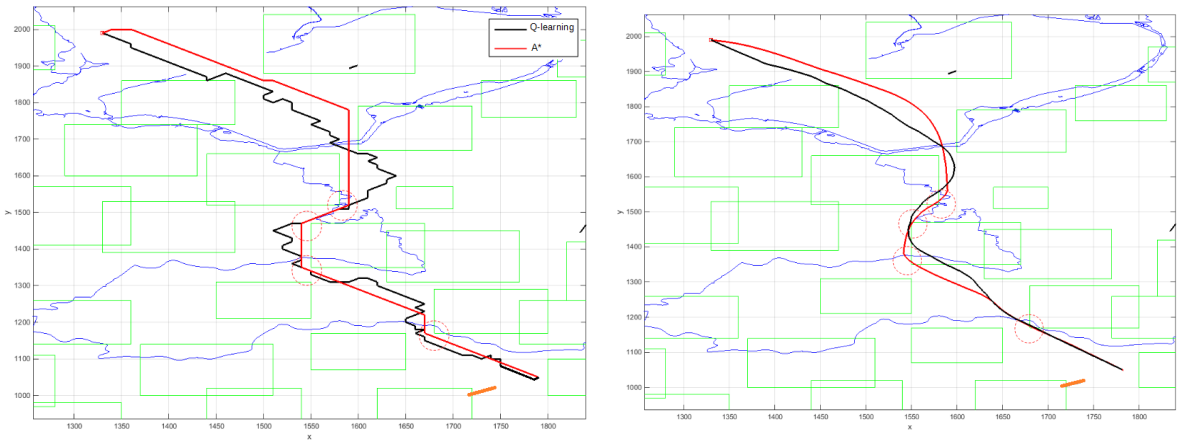
(a) Target trajectories

(b) Smoothed trajectories

Figure 5.2: Woensdrecht route

The airports can be sorted into two categories. The ones that are surrounded by population clusters and the airports that are not surrounded. The trajectories to the latter will be discussed first.

In this scenario, Q-learning trajectories to runways that are not surrounded by population clusters vary in length between 94% and 112% of the A* trajectories. The fact that the A* algorithm creates shorter paths is in agreement with the theory introduced in section 3.3. It must, however, also be noted that even though the target trajectory of the A* algorithm circumvents the obstacles, the dynamic model trajectory may not.
This can be explained by the fact that the A* algorithm, to circumvent an obstacle, has to create routes sometimes right past the borders of the obstacle. The dynamic model creates a path in such a way that it fits the target trajectory optimally with a smooth line. If then as indicated in figure 5.3a two consecutive steps in the A* target trajectory make a 45 degree angle at one of the vertices of an obstacle, the resulting path shown in figure 5.3b will be generated over an obstacle.

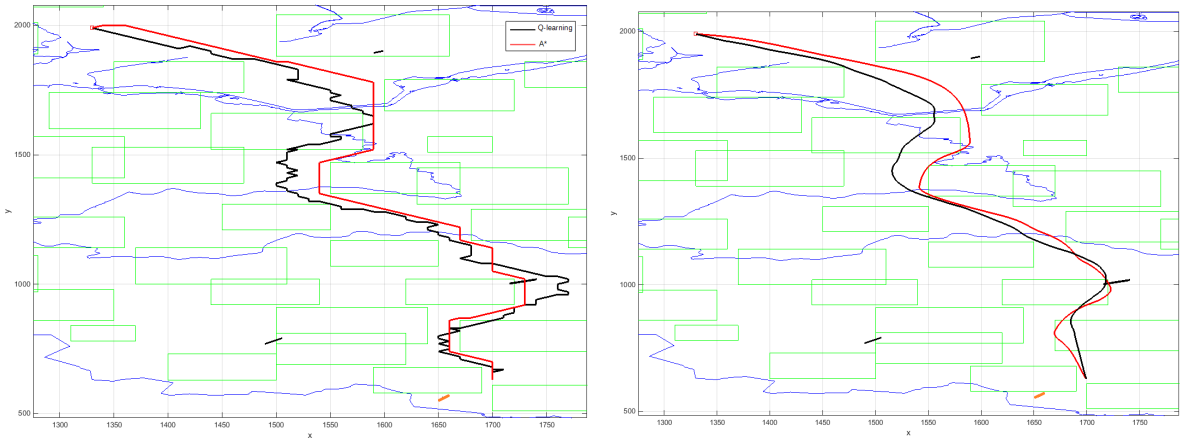


(a) Target trajectory

(b) Smoothed trajectory

Figure 5.3: Volkel route

The result that Q-learning can create a shorter trajectory is due to the fact that, as noted in section 4.1, it can create trajectories through obstacles. The Q-learning algorithm balances between a longer route resulting in a higher summed distance cost or a shorter route but receiving penalties. An example of this is shown in figure 5.4a in which can be seen that the Q-learning trajectory to runway 21 at Kempen airport is generated through the outskirt of an obstacle while the A* algorithm circumvents all obstacles.



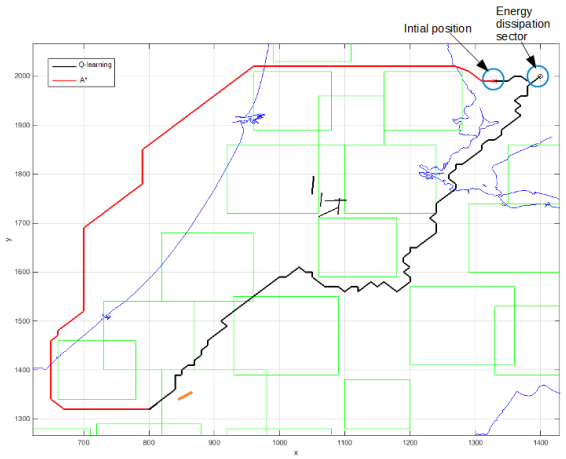
(a) Target trajectory

(b) Smoothed trajectory

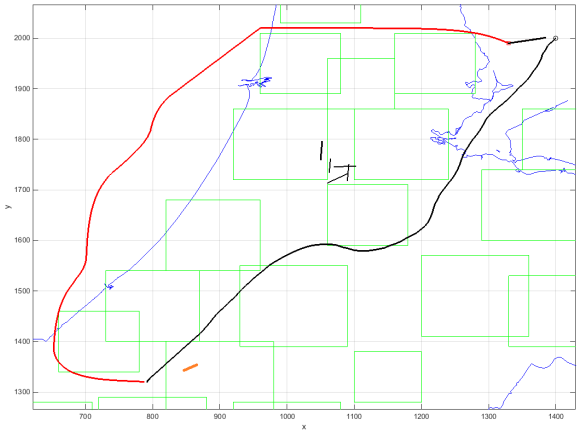
Figure 5.4: Kempen route

The horizontal track length of the Q-learning trajectories to airports that are surrounded by population clusters, however, vary between 55% and 74% compared to the A* algorithm. In the target trajectories generated

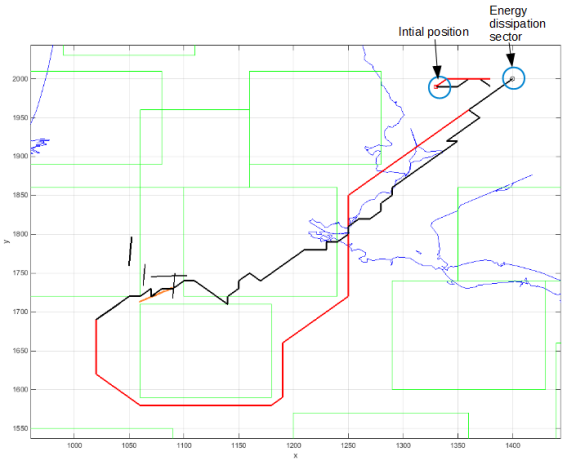
between the aircraft initial position and airports of Rotterdam, Schiphol and Gilze Rijen shown in respectively figures 5.5a, 5.5c and 5.5e and the pertaining smoothed trajectory shown in figures 5.5b, 5.5d and 5.5f it can be seen that the presence of the hard constraints, earlier mentioned in this chapter, result in longer A* target trajectories compared with the Q-learning trajectories. The airports are surrounded by clusters of obstacles, which the A* algorithm has to circumvent. The Q-learning algorithm however, can create trajectories through the obstacles creating shorter routes.



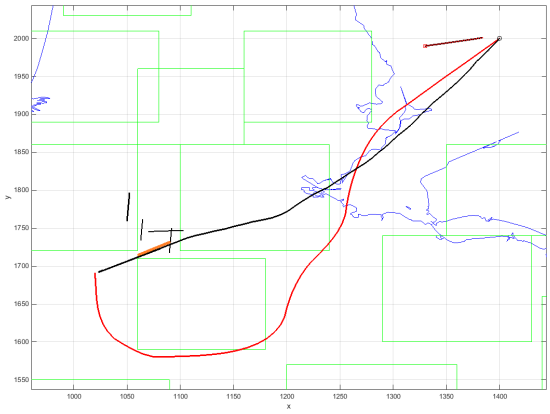
(a) Rotterdam target trajectory



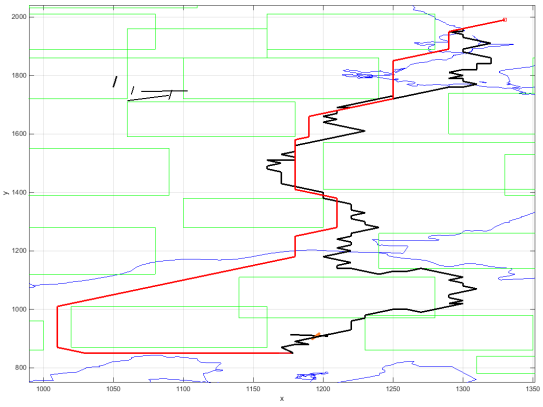
(b) Rotterdam smoothed trajectory



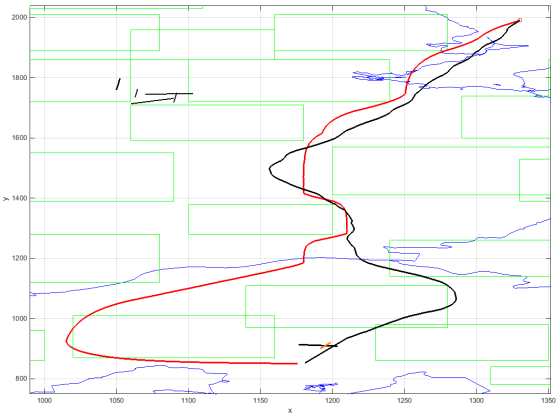
(c) Schiphol target trajectory



(d) Schiphol smoothed trajectory



(e) Gilze Rijen target trajectory



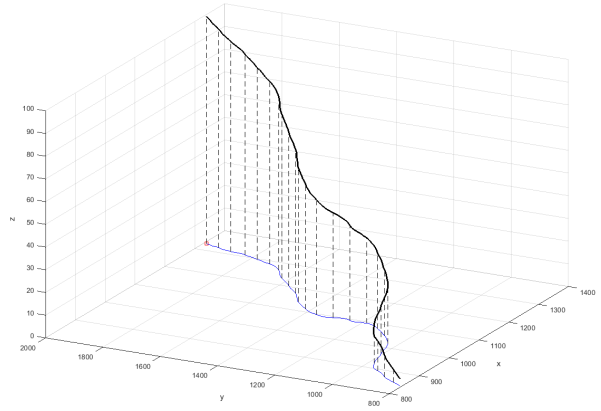
(f) Gilze Rijen smoothed trajectory

Figure 5.5: Airport routes

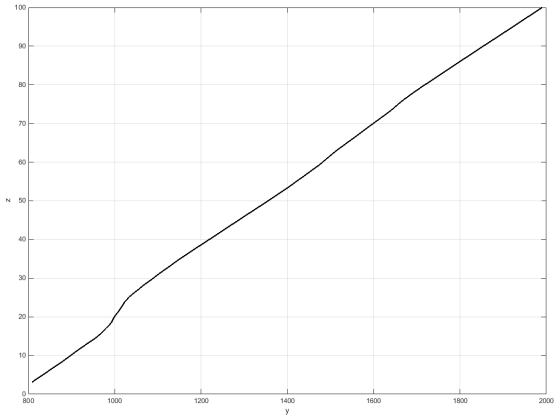
5.2. Vertical Trajectories

The trajectory to runway 25 at Woensdrecht airbase has a descent angle which satisfies the flight path angle constraint defined in section 2.2.2 and therefore the trajectory between the initial aircraft position and the airport is a direct route. In figure 5.6a the 3D trajectory of the Woensdrecht route is displayed. In this figure, the black line is the 3D trajectory while the blue line indicates the ground track.
In the vertical trajectory, shown in figure 5.6b, it can be seen that the trajectory starts at an altitude of 10 km and ends at the target point at which the aircraft must be stabilized on the glide slope, indicated in figure 2.6. The trajectory is feasible because it satisfies all the constraints.

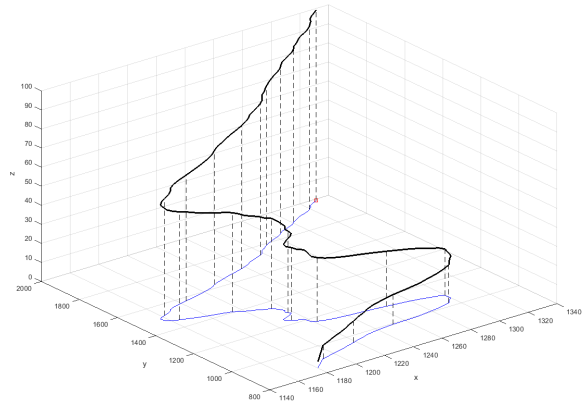
Another runway to which a direct route can be made from the initial aircraft position is, runway 02 at Gilze Rijen airbase. The target trajectory for this route is displayed in figure 5.5e and resulting smoothed trajectory in figure 5.5f. The 3D and vertical trajectory for this route are shown in respectively figures 5.6c and 5.6d.



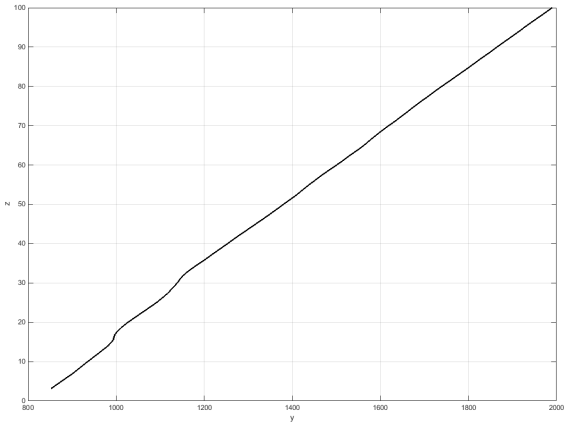
(a) Woensdrecht 3D trajectory



(b) Woensdrecht vertical trajectory



(c) Gilze Rijen 3D trajectory

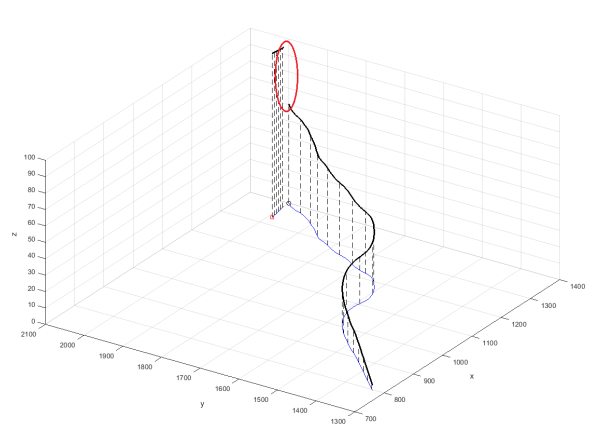


(d) Gilze Rijen vertical trajectory

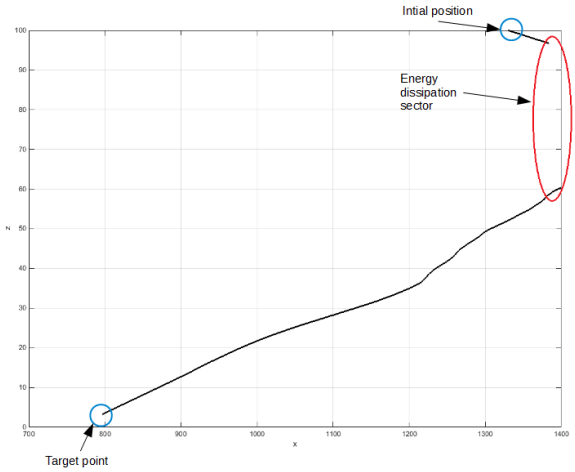
Figure 5.6: Direct routes

A direct Q-learning trajectory between the initial position and the target point of runway 06 at Rotterdam airport is unfeasible because the required angle of descent, resulting from the smoothed horizontal and vertical distance, is too steep and does not satisfy the descent angle constraint. The trajectories are therefore generated in two segments. In figures 5.5a and 5.5b it can be seen that the first segment is a trajectory from the initial aircraft position until the energy dissipation sector. The second segment is between the energy dissipation sector and the target point of the reachable airport. These two segments can also be seen in figure 5.7a and the corresponding vertical plane in figure 5.7b. Between the two segments, an altitude difference of 4 kilometers is indicated. This vertical distance has to be dissipated in the energy dissipation sector.

If the initial position is closer to a reachable airport, more altitude difference has to be dissipated in one of the energy dissipation sectors. In figure 5.5c and figure 5.5d respectively the target trajectory and the smoothed trajectory to runway 06 (Kaagbaan) at Schiphol airport are shown. It can be seen that the trajectory is divided in two segments. From the initial position a trajectory is generated to the dissipation sector. The second segment is generated between the dissipation sector and the airport. These two segments can also be seen in the 3D trajectory in figure 5.8a and the vertical plane in figure 5.8b. This airport is closer to the initial aircraft position than the airport of Rotterdam. This can also be deduced from the bigger altitude difference compared to figure 5.7b that has to be dissipated in order to create a feasible trajectory.

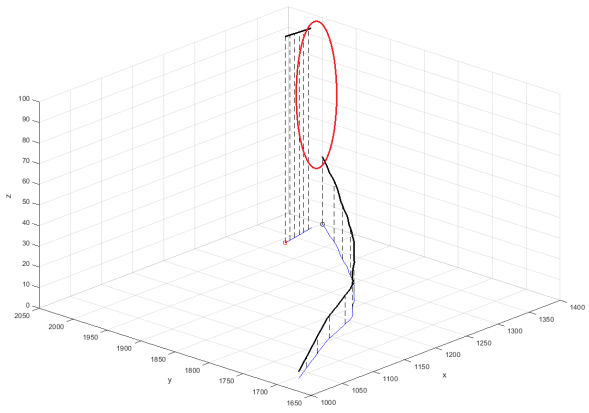


(a) 3D trajectory

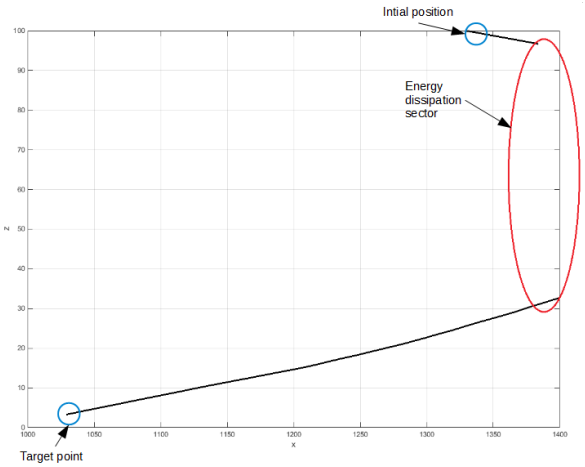


(b) Vertical trajectory

Figure 5.7: Rotterdam route



(a) 3D trajectory



(b) Vertical trajectory

Figure 5.8: Schiphol route

5.3. Ranking

After the trajectories are generated by the emergency trajectory planner, they are ranked. In section 2.4 it is explained that this is done using equation 5.1 which is adapted from Atkins et al. (2006):

$$U = \sum_i C_i \cdot w_i = C_1 \cdot \frac{r_1}{r_{1,max}} + C_2 \cdot \frac{r_w}{r_{w,max}} + C_3 \cdot q_I + C_4 \cdot (\frac{d}{d_{max}}) + C_5 \cdot q_s + C_6 \cdot q_f, \tag{5.1}$$

where r_l is the runway length, r_w is the runway width, q_I instrument approach quality, distance d from the footprint boundary, surface quality q_s and facility availability measure q_f. The values of r_l,r_w and d are normalized [0.0 1.0]. The weighting factors have to be determined by the respective experts; this could be airlines, air traffic controllers or pilots. in this research the weighting factors determined by Atkins et al. (2006) are used which are set to $\{C_1, C_2, C_3, C_4, C_5, C_6\}$= $\{0.15, 0.15, 0.15, 0.15, 0.1, 0.1\}$.

The value of U determines the rank of each created trajectory. The values of r_l,r_w and d are normalized and are different for each initial conditions. The values for this situation are $r_{l_{max}}$ = 3800, $r_{w_{max}}$ = 60, d_{max} = 139.25. The values of q_I, q_s are fixed and depend on the facilities at the airport. In this research the values of q_I, q_s determined by Atkins et al. (2006) are used and are listed in figure 5.9.

Description (max value used)	Value
Instrument approach (q_I)	
WAAS, ILS/MLS	1.0
LOC with RWY designation	0.95
LOC w/o RWY designation	0.85
LDA w/RWY designation	0.8
LDA w/o RWY designation	0.7
GPS, LORAN, RNAV w/RWY	0.6
VOR, NDB, SDF w/RWY	0.5
GPS, LORAN, RNAV, VOR, NDB, SDF w/o RWY	0.2
Runway surface (q_S)	
Asphalt	1.0
Concrete	1.0
Metal, brick, etc. (VTOL)	0.5
Wood	0.2
Turf/gravel/dirt	0.1
Airport facilities (q_f)	
Fuel of required type (Jet-A)	0.25
Airframe maintenance	
Major	0.25
Minor	0.125
Power plant maint.	
Major	0.25
Minor	0.125
Bulk oxygen	0.25

WAAS = wide area augmentation system, ILS = instrument landing system, MLS = microwave landing system, LOC = localizer, RWY = runway, LDA = localizer type directional aid, GPS = global positioning system, LORAN = long range navigation, RNAV = area navigation, VOR = very high frequency omni-directional range, NDB = nondirectional beacon, SDF = simplified directional facility, and VTOL = vertical takeoff and landing.

Figure 5.9: Quality measures for runway utility computation (Atkins et al., 2006)

The ranked runways are shown in table 5.1. From the table it can be deduced that the runways of Schiphol have the first 6 positions in the rank. This is due to the fact that it has the longest runways, it is, after Lelystad, the furthest away from the footprint boundary and it has the highest quality measures. The pilots can choose to land on one of the runways of Schiphol if they choose to follow ranking, however a situation can occur in which Schiphol closes its runway even for emergencies. The pilots can then choose between one of the 18 remaining options. Depending on the problem with the aircraft, either an airport with better facilities might be chosen to repair the malfunctioning parts of the aircraft or the nearest airport. By giving a ranking of runways and also notifying where this ranking is based on, gives the pilot the opportunity to bring the emergency situation to a good end.

Rank	Airport	Runway	r_l	r_w	q_l	q_s	q_f	d (km)	U
1	Schiphol	Polderbaan	3800	60	1	1	1	131	0.79
2	Schiphol	Kaagbaan	3500	45	1	1	1	131	0.74
3	Schiphol	Buitenveldertbaan	3450	45	1	1	1	131	0.74
4	Schiphol	Aalsmeerderbaan	3400	45	1	1	1	131	0.74
5	Schiphol	Zwanenburgbaan	3300	45	1	1	1	131	0.74
6	Schiphol	Oostbaan	2015	45	1	1	1	131	0.68
7	Leeuwarden	06/24	2957	50	1	1	0.25	80.66	0.60
8	Rotterdam	06/24	2200	45	1	1	0.375	86.17	0.58
9	Volkel	06L/24R	3024	45	1	1	0.25	61.31	0.57
10	Woensdrecht	07/25	2440	45	1	1	0.75	33.65	0.57
11	Leeuwarden	09/27	1999	50	1	1	0.25	80.66	0.57
12	Gilze Rijen	10/28	2779	45	1	1	0.25	58.26	0.56
13	Eindhoven	03/21	3000	45	1	1	0.25	43.89	0.55
14	Groningen	05/23	2500	45	1	1	0.375	50.56	0.55
15	De Kooy	03/21	1275	30	1	1	0.25	118.29	0.53
16	Volkel	06R/24L	3027	23	1	1	0.25	61.31	0.52
17	Kempen	03/21	2750	45	1	1	0.25	20.40	0.52
18	Lelystad	05/23	1250	30	1	0.5	0.25	139.25	0.50
19	Enschede	05/23	2987	45	1	0.2	0.25	44.64	0.47
20	Lt. Gen. Best	06/24	2988	45	1	0.2	0.25	43.83	0.47
21	Gilze Rijen	02/20	1996	30	1	0.8	0.25	58.26	0.47
22	Groningen	01/19	1500	45	1	0.5	0.375	50.56	0.46
23	Deelen	02/20	2400	50	0.5	0.2	0.25	92.74	0.44
24	Teuge	08/26	1199	27	1	0.2	0	96.67	0.39

Table 5.1: Ranked runways

6

Conclusions & Recommendations

6.1. Conclusions

The objective of this research was:

To develop an emergency trajectory planner that can generate routes taking into account loss of life.

The examination of literature on this subject led to the formulation of the following research question:

How can an emergency trajectory planner generate gliding trajectories using Approximate Dynamic Programming to safely land a transport aircraft with total loss of thrust?

When the emergency situation of an aircraft starts in which it looses all thrust, a footprint is generated. Within this footprint of which the diameter is determined by the maximum straight glide distance, all runways are identified. To each runway a path has to be created. In robotics, Approximate Dynamic Programming is used to create paths. By moving around, the robot explores the environment and tries to create a path through it. The reward function gives penalties when obstacles are encountered and a reward when the goal state is found. The approximate dynamic programming algorithms that were discussed in chapter 3 were Sarsa, Q-learning and their variants with eligibility traces Sarsa λ and Q-λ. By using eligibility traces the algorithm is multiple times slower because at every step it has to update the trace for every state in the value function. If the environment is unknown the use of these traces can speed up finding the optimal solution which compensates for the longer processing time. In this research however trajectories have to be generated in a short amount of time in a known environment making Sarsa λ and Q-λ unsuitable for the emergency trajectory planner.

The remaining ADP algorithms, Sarsa and Q-learning were compared with each other in two test scenarios using the A* algorithm as the benchmark. The A* algorithm was chosen as the benchmark because the trajectories generated give the optimal route while avoiding all obstacles if certain conditions are met, which were defined in section 3.3.1.
These two scenarios were set in the same environment consisting out of multiple small obstacles. In theory, Sarsa and Q-learning have a probability of 1 of finding the optimal policy, however this condition consists of visiting all states an unlimited amount of times. This is not very practical and therefore to converge the algorithm to a solution a stopping criterion was implemented. To calculate the value function and its optimal policy an ϵ-greedy policy was used. The two variables that have a direct influence on the probability of finding an optimal solution are learning rate α and exploration probability ϵ. These two variables were tested for Sarsa and Q-learning for a range of values to determine which algorithm produces the best results. It was concluded that Q-learning produced the best results, however by using an ϵ-greedy policy to explore the environment and a stopping condition to converge, every trajectory generated for the same route is different. The simulation time of Q-learning and A* in the two scenarios were comparable.

In chapter 5 the trajectories generated by the emergency trajectory planner from the initial aircraft position to the reachable airports for a scenario in which the aircraft looses all trust were analyzed and compared with the A* algorithm trajectories. From the results it could be concluded that the Q-learning algorithm creates shorter trajectories to airports which are surrounded by obstacle clusters than the A* trajectories. As stated in section 4.2, Q-learning can balance between flying over a populated area and receive penalties, effectively increasing the risk of loss of life on ground, or flying a longer route around the obstacle, effectively increasing the risk of loosing control of the aircraft by having to be airborne for a longer period of time. The A* algorithm avoids all obstacles and therefore has to generate a path around the population clusters, resulting in longer trajectories.

On routes without population clusters the Q-learning trajectories for this specific scenario varied between 94% and 112% compared with the A* trajectories. The in comparison shorter trajectories are explained by the fact that the Q-learning trajectories can be generated through obstacles, as stated before. The longer paths are due to the fact that generated Q-learning trajectories are not optimal in comparison with the A* trajectories. It must however, also be noted that although the target trajectories of the A* algorithm do not go through obstacles the smoothed paths created by the dynamic model are in fact able to do the opposite. This is because the target trajectories do not take into account flight dynamics when avoiding obstacles. The smoothed trajectory based on the target trajectory therefore can be generated through obstacles. In contrary to the two test scenarios, the simulation time of Q-learning for generating trajectories to the reachable runways was multiple times higher for all trajectories in comparison with the A* algorithm.

Although approximate dynamic programming is a valuable algorithm for path planning in robotics, it proves less suitable to create trajectories on line in emergency situations. While it successfully generated trajectories to the reachable runways, the time the algorithm needs to create these trajectories is too long to be able to implement it in emergency situations. By using a stopping criterion to converge the value function, optimality of the results is not guaranteed which is an undesirable property in emergency situations. Also by not taking into account flight dynamics when generating the trajectory, obstacles might not be avoided optimally.

6.2. Recommendations

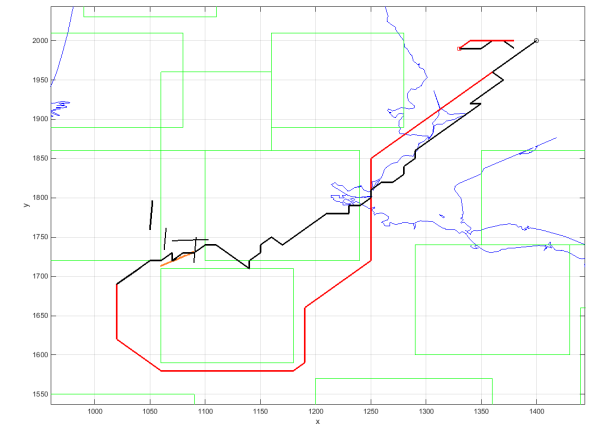
Derived from the conclusions and drawbacks found in this research, the following points can be addressed to improve the performance of the emergency trajectory planner:

- Explore the possibility of reconfiguring the A* function to include soft constraints. By doing so the problem of having to generate long trajectories when encountering obstacle clusters can be solved.
- Investigate the possibility of taking into account heading angle and flight dynamics when generating target trajectories.
- Add an element to the utility function which ranks the airports to account for trajectories generated over populated areas. Airports that can only be reached by going through obstacles have a higher risk of loss of life on ground and should therefore be ranked lower.
- Investigate the possibility of creating obstacles to represent bad weather areas resulting in trajectories circumventing areas with bad weather.
- Explore other ways of dissipating energy. If an aircraft is flying over land and in the area no energy dissipation sectors can be found it must loose altitude and speed using an alternative way.

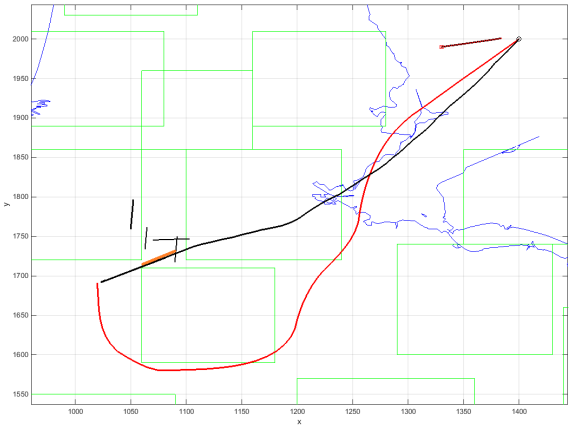
A

Appendix

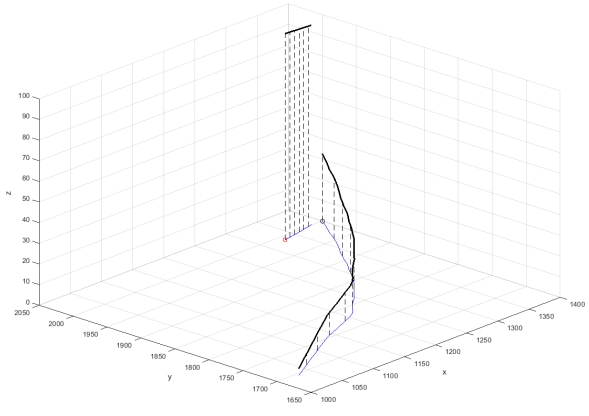
A.1. Trajectory results



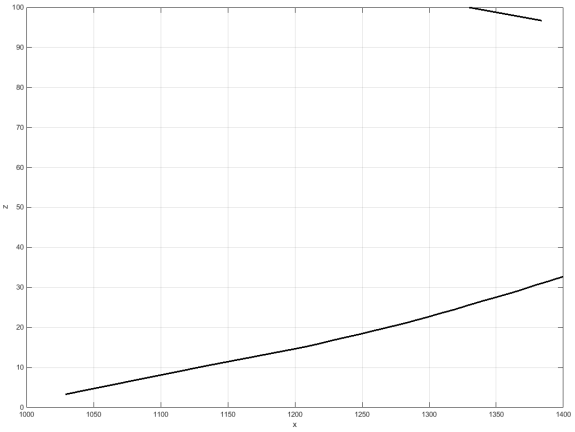
(a) Target trajectory



(b) Smoothed trajectory

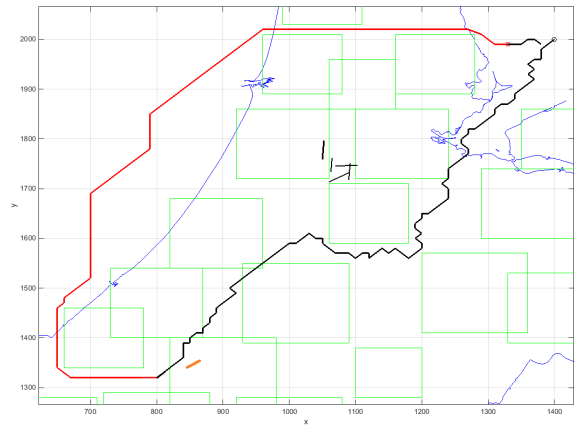


(c) 3D trajectory

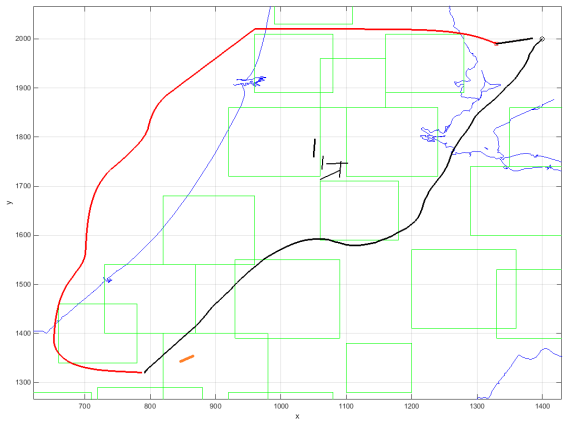


(d) Vertical trajectory

Figure A.1: Schiphol route

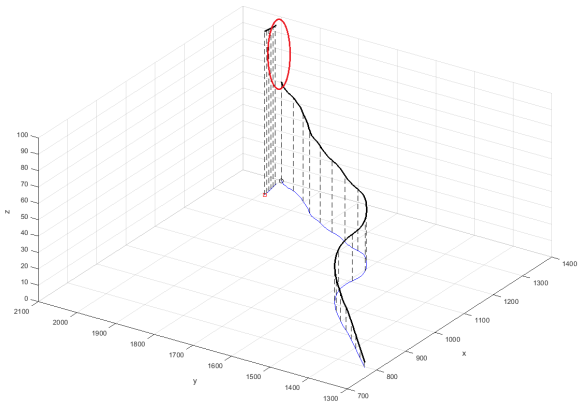


(a) Target trajectory

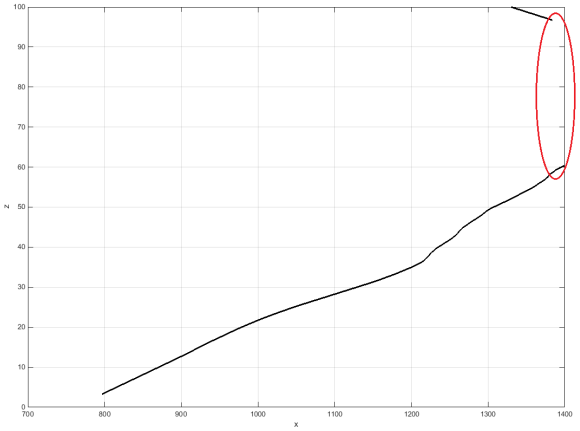


(b) Smoothed trajectory

Figure A.2: Rotterdam horizontal route

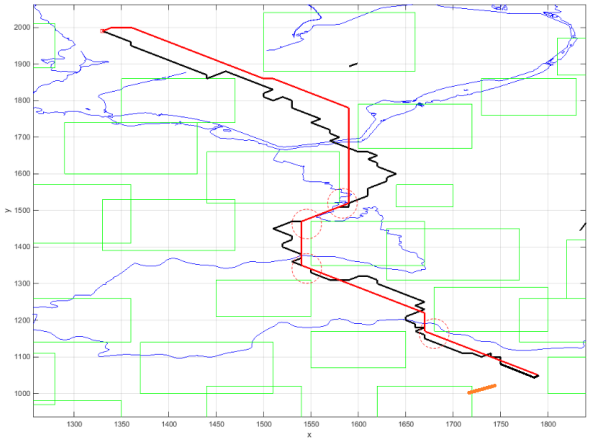


(a) 3D trajectory

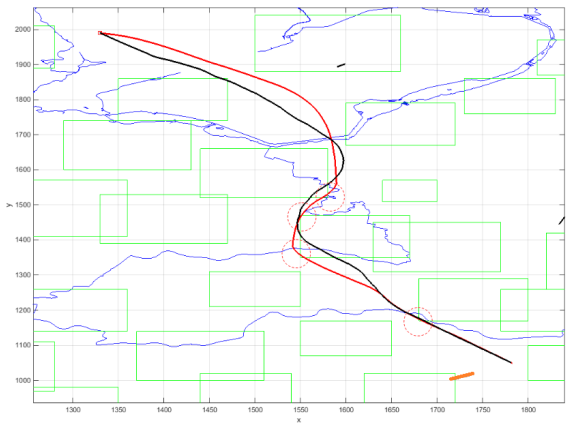


(b) Vertical trajectory

Figure A.3: Rotterdam vertical route

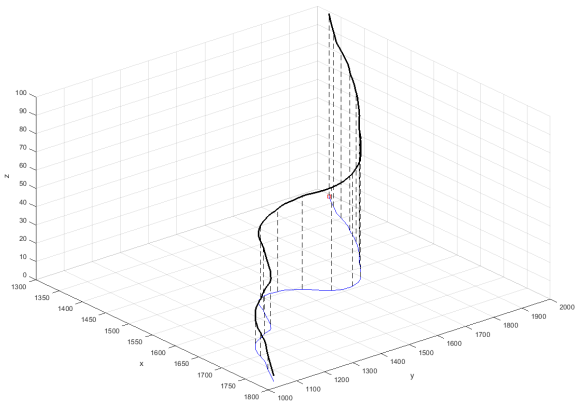


(a) Target trajectory

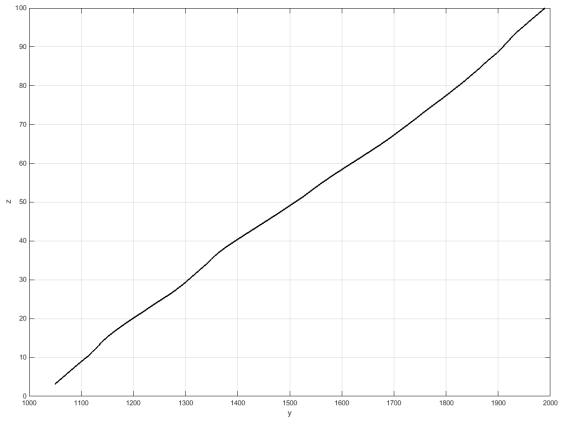


(b) Smoothed trajectory

Figure A.4: Volkel horizontal route

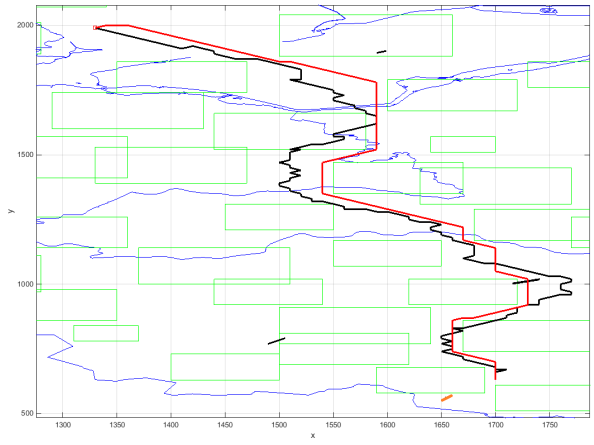


(a) 3D trajectory

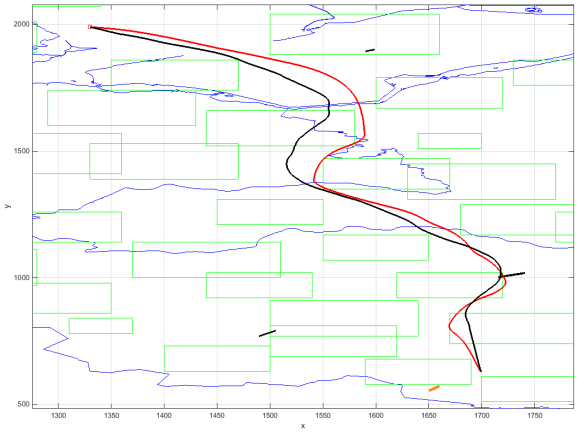


(b) Vertical trajectory

Figure A.5: Volkel vertical route

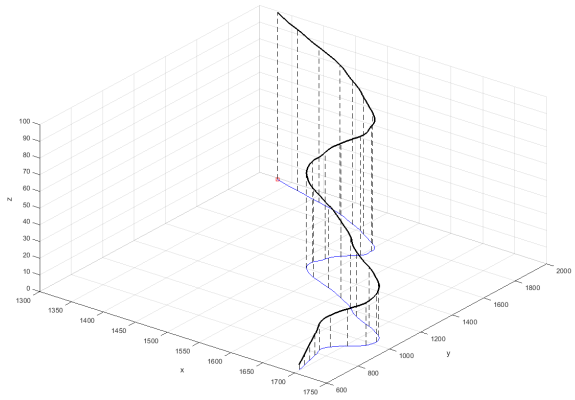


(a) Target trajectory

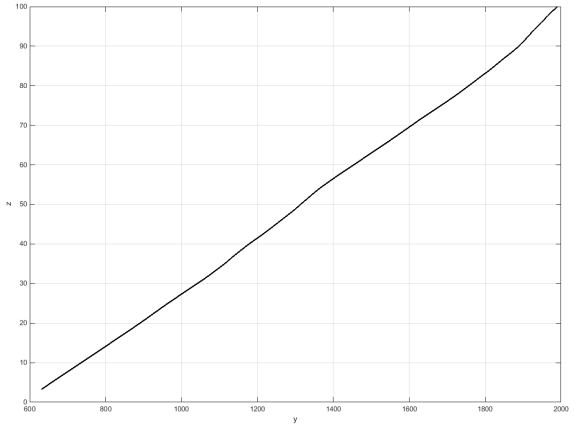


(b) Smoothed trajectory

Figure A.6: Kempen horizontal route

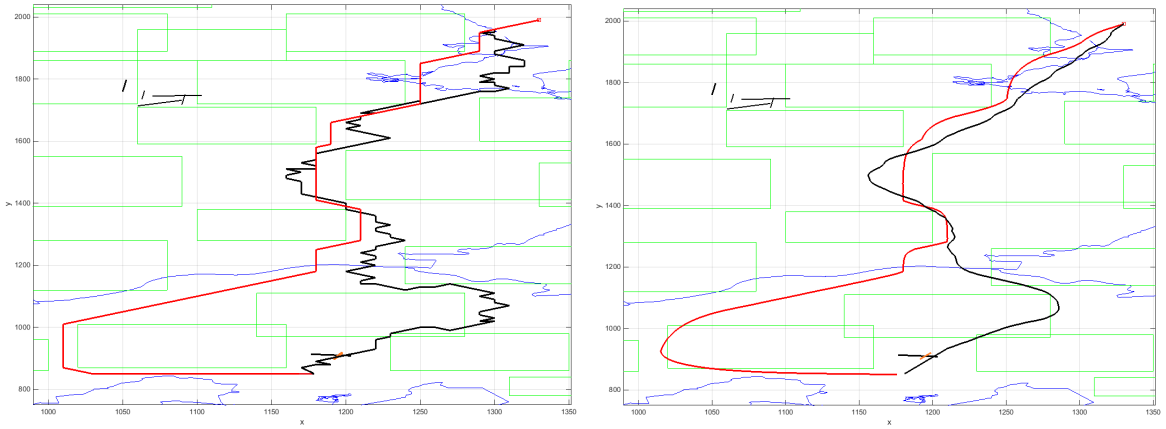


(a) 3D trajectory



(b) Vertical trajectory

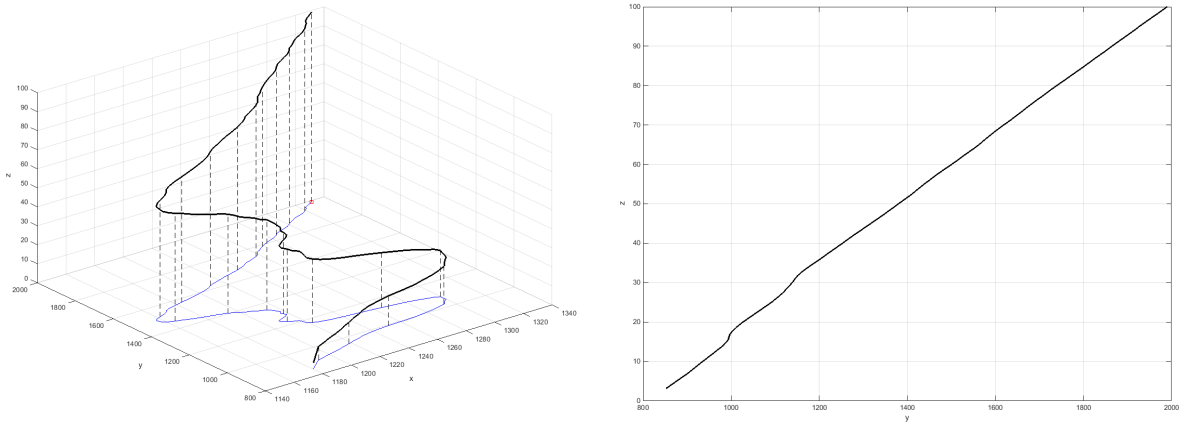
Figure A.7: Kempen vertical route



(a) Target trajectory

(b) Smoothed trajectory

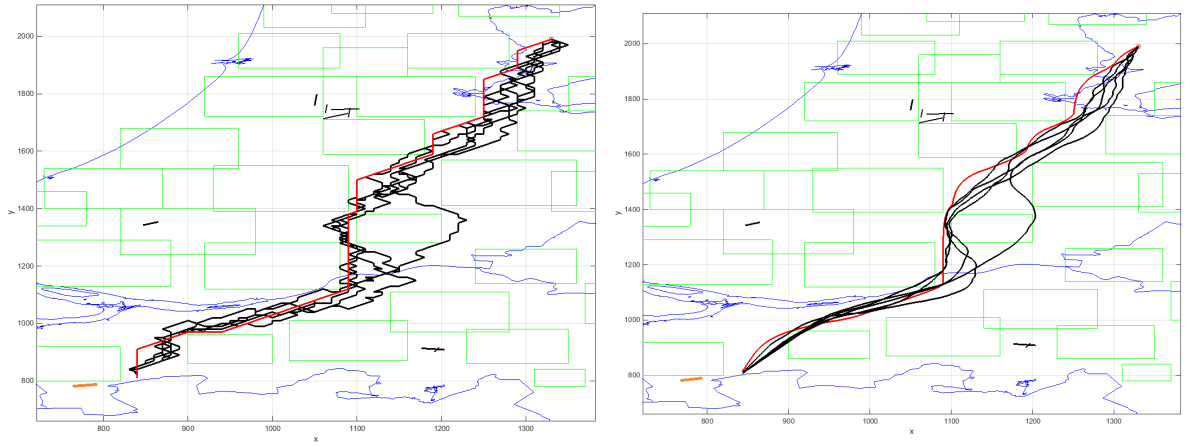
Figure A.8: Gilze Rijen horizontal route



(a) 3D trajectory

(b) Vertical trajectory

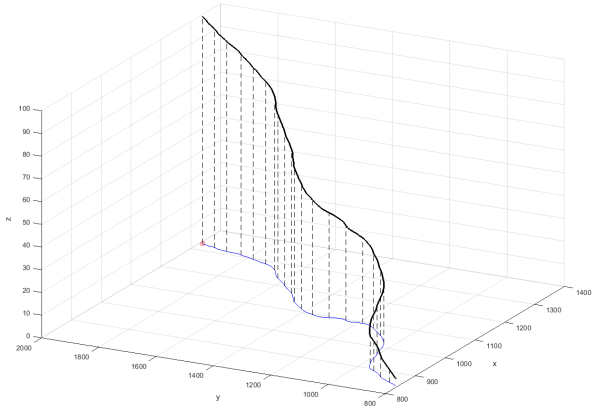
Figure A.9: Gilze Rijen vertical route



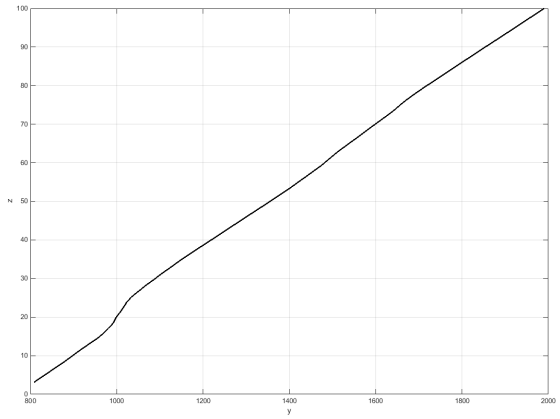
(a) Target trajectory

(b) Smoothed trajectory

Figure A.10: Woensdrecht horizontal route



(a) 3D trajectory



(b) Vertical trajectory

Figure A.11: Woensdrecht vertical route

A.2. Airports

A.2.1. Schiphol (EHAM)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
Polderbaan	3800	60	1	1	1
Kaagbaan	3500	45	1	1	1
Buitenveldertbaan	3450	45	1	1	1
Aalsmeerbaan	3400	45	1	1	1
Zwanenburgbaan	3300	45	1	1	1
Oostbaan	2014	45	1	1	1

Table A.1: Schiphol Airport runways

A.2.2. Teuge Airport (EHTE)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
08/26	1199	27	1	0.2	0

Table A.2: Teuge Airport runway

A.2.3. Maastricht Aachen Airport (EHBK)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
03/21	2750	45	1	1	0.25

Table A.3: Maastricht Aachen Airport runway

A.2.4. Kempen Airport (EHBD)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
03/21	1200	23	1	0.2	0

Table A.4: Kempen Airport runway

A.2.5. De Kooy Airfield (EHKD)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
03/21	1275	30	1	1	0.25

Table A.5: De Kooy Airfield runway

A.2.6. Eindhoven Airport (EHEH)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
03/21	3000	45	1	1	0.25

Table A.6: Eindhoven Airport runway

A.2.7. Enschede Airport Twente (EHTW)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
05/23	2987	45	1	0.2	0.25

Table A.7: Enschede Airport Twente runway

A.2.8. Groningen Airport Eelde (EHGG)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
05/23	2500	45	1	1	0.375
01/19	1500	45	1	0.5	0.375

Table A.8: Groningen Airport Eelde runways

A.2.9. Lelystad Airport (EHLE)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
05/23	1250	30	1	0.5	0.25

Table A.9: Lelystad Airport runway

A.2.10. Rotterdam The Hague Airport (EHRD)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
06/24	2200	45	1	1	0.375

Table A.10: Rotterdam The Hague Airport runway

A.2.11. Deelen Air Base (EHDL)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
02/20	2400	50	0.5	0.2	0.25

Table A.11: Deelen Air Base runway

A.2.12. Gilze-Rijen Air Base (EHGR)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
10/28	2779	45	1	1	0.25
02/20	1996	30	1	0.8	0.25

Table A.12: Gilze-Rijen Air Base Runways

A.2.13. Leeuwarden Air Base (EHLW)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
06/24	2957	50	1	1	0.25
09/27	1999	50	1	1	0.25

Table A.13: Leeuwarden Air Base runways

A.2.14. Volkel Air Base (EHVK)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
06L/24R	3024	45	1	1	0.25
06R/24L	3027	23	1	1	0.25

Table A.14: Volkel Air Base runways

A.2.15. Lieutenant General Best Barracks (EHDP)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
06/24	2988	45	1	0.2	0.25

Table A.15: Lieutenant General Best Barracks runway

A.2.16. Woensdrecht Airbase (EHWO)

Runway	Length r_l [m]	Width r_w [m]	Runway surface (q_S) [-]	Instrument approach (q_I) [-]	Airport facilities (q_f) [-]
07/25	2440	45	1	1	0.75

Table A.16: Woensdrecht Airbase runway

A.3. Atmospheric Parameters

Parameter	Equation
Air Temperature ($°K$) Static Pressure ($\frac{N}{m^2}$) Pressure Ratio Air Density ($\frac{kg}{m^3}$) Density Ratio	$T_{ISA} = T_{ISA_0} - 6.5\frac{h}{1000}$ $P = P_0(1 - 0.065\left(\frac{h}{288.15}\right)^5.2561$ $PR = \frac{P}{P_0}$ $\rho = \frac{P}{287.04 \cdot T_{ISA}}$ $\delta = \frac{\rho}{\rho_0}$
$T_{ISA,0} = 288.15°K$ $P_0 = 101325\frac{N}{m^2}$ $\rho_0 = 1.225\frac{kg}{m^3}$	

Table A.17: Atmospheric parameters

Bibliography

A. Adler, A. Bar-Gill, and N. Shimkin. Optimal Flight Paths for Engine-Out Emergency Landing. *2012 24th Chinese Control and Decision Conference (CCDC)*, 2012.

AOPA. General Aviation Accidents in 2012. *24th Joseph T.Nall Report*, 2015.

D. B. Aranibar and P. J. Alsina. Reinforcement Learning-Based Path Planning for Autonomous Robots. *EnRI-XXIV Congresso da Sociedade Brasileira de Computacao*, page 10, 2004.

E. M. Atkins, I. Portillo Alonso, and M. J. Strube. Emergency Flight Planning Applied to Total Loss of Thrust. *Journal of Aircraft*, 43(4):1205–1216, July 2006.

J. T. Betts. Survey of Numerical Methods for Trajectory Optimization. *Journal of Guidance, Control, and Dynamics*, 21(2):193–207, 1998. ISSN 0731-5090. doi: 10.2514/2.4231. URL http://arc.aiaa.org/doi/abs/10.2514/2.4231.

Boeing. Statistical Summary of Commercial Jet Airplane Accidents: Worldwide Operations | 1959-2014. http://www.boeing.com/resources/boeingdotcom/company/about_bca/pdf/statsum.pdf, 2014. Accesed on: 1-12-2015.

J. D. Boskovic and R. K. Mehra. An Integrated Fault Management System for Unmanned Aerial Vehicles. 2003.

C. Brady. *The Boeing 737 Technical Guide*. Tech Pilot Services Ltd, 2015.

K. Brinkman and H.G. Visser. Optimal Turn-Back Maneuver after Engine Failure in a Single-Engine Aircraft during Climb-Out. *45th AIAA Aerospace Sciences Meeting and Exhibit*, 2007.

L. Busoniu, R. Babuska, B. de Schutter, and D. Ernst. *Reinforment Learning and Dynamic Programming Using Function Approximators*. CRC Press, 2010a.

L. Busoniu, B. de Schutter, and R. Babuska. Approximate dynamic programming and reinforcement learing. *Studies in Computational Intelligence*, 281:3–44, 2010b.

P.V.C. Caironi and M. Dorigo. Training and Delayed Reinforcements in Q-Learning Agents. *Tech. Rep. IRIDIA Universite de Bruxelles*, 1994.

CBS. CBS gebiedsindelingen. http://www.nationaalgeoregister.nl, 2015. Accesed on: 1-12-2015.

T. L. Chen and A. R. Pritchett. Development and Evaluation of a Cockpit Decision-Aid for Emergency Trajectory Generation Introduction. *Journal of Aircraft*, 38(5), 2001.

M. Coombes, W. Chen, and P. Render. Reachability analysis of landing sites for forced landing of a UAS. *2013 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 542–549, May 2013.

W. Dabney. *Adaptive Step-sizes for Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 2014.

R. Fernandes de Oliveira and C. Büskens. Emergency flight replanning for minimum loss of life risk using a decoupled trajectory optimization approach. *2013 Aviation Technology, Integration, and Operations Conference*, pages 1–14, August 2013.

(Flight Safety Foundation FSF. Accident description. http://aviation-safety.net/database/record.php?id=19830723-0, 2015. Accesed on: 1-12-2015.

D. Garg. *Advances in Global Pseudospectral Methods for Optimal Control*. PhD thesis, University of Florida, 2011.

A. Gosavi. *Simulation-Based Optimization: Parametric Optimization Techniques and Re-inforcement Learning*. Springer, 2014.

P. K. Goswami, I.and Das, A. Konar, and R Janarthanan. Extended Q-Learning Algorithm for Path-Planning of a Mobile Robot. *Simulated Evolution and Learning*, 6457:379–383, 2010.

GPIAA. Accident investigation final report:all engines-out landing due to fuel exhaustion air transat airbus a330-243 marks c-gits lajes, azores, portugal 24 august 2001, 2001.

M. Grzes. *Improving Exploration in Reinforcement Learning through Domain Knowledge and Parameter Analysis*. PhD thesis, The University of York, 2010.

P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

J. Hoffren and T. Raivio. Optimal maneuvering after engine failure. *AIAA*, 2000.

A. K. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters 31*, pages 651–666, 2010.

B. W. Jett. The Feasibility of Turnback from a Low Altitude Engine Failure During the Take- off Climb-out Phase. *AIAA 20th Aerospace Sciences Meeting*, 1982.

R. Kala, A. Shukla, and R. Tiwari. Robot Path Planning using Dynamic Programming with Accelerating Nodes. *Paladyn, Journal of Behavioral Robotics*, 3(1):23–34, 2012. ISSN 2081-4836. doi: 10.2478/s13230-012-0013-4.

L. Kallenberg. Markov decision processes, 2007.

S. Koenig and R. G. Simmons. The effect of representation and knowledge on goal-directed exploration with reinforcement-learning algorithms. *Machine Learning*, 22(1-3):227–250, 1996. ISSN 0885-6125. doi: 10.1007/BF00114729.

A. Konar, I. G. Chakraborty, S. J. Singh, L.C. Jain, and A. K. Nagar.

R.E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, March 1990.

F. Kunz. An Introduction to Temporal Difference Learning. *Seminar on Autonomous Learning Systems*, 2013.

H. A. Lazos Fernandez. Noise Abatement 3D Path Planning Using a Snake Algorithm. 2015.

K. Macek, I. Petrovic, and N. Peric. A reinforcement learning approach to obstacle avoidance of mobile robots. *7th International Workshop on Advanced Motion Control. Proceedings (Cat. No.02TH8623)*, pages 462–466. doi: 10.1109/AMC.2002.1026964.

N. Meuleau, C. Plaunt, D. E. Smith, and T Smith. An Emergency Landing Planner for Damaged Aircraft. *Proceedings of the 21 Innovative Applications of Artificial Intelligence Conference*, 2009.

R. Ortner. Online regret bounds for Markov decision processes with deterministic transitions. *Theoretical Computer Science*, 411:2684–2695, 2010.

T. Peng, Z. Shuguang, J. Lei, and T. Zhi. A Novel Emergency Flight Path Planning Strategy for Civil Airplanes in Total Loss of Thrust. *Procedia Engineering*, 17:226–235, January 2011.

W. B. Powell. *Approximate Dynamic Programming, Solving the Curses of Dimensionality*. Wiley, 2007.

D. Precup, R. S. Sutton, and S. Dasgupta. Off-policy temporal-difference learning with function approximation. *Proceedings of the Eigh- teenth International Conference on Machine Learning*, pages 417–424, 2001.

D. F. Rogers. The Possible ‘ Impossible ’ Turn. *AIAA Journal of Aircraft*, 32:392–397, 1995.

C. S. Sallaberger. Optimal Robotic Path Planning Using Dynamic Programming. *Acta Astronautica*, 35(2/3): 143–156, 1995.

M. Santos, J.A. Martin H, V. Lopez, and G. Botella. Dyna-H: A heuristic planning reinforcement learning algorithm applied to role-playing game strategy decision systems. *Knowledge-Based Systems*, 32:28–36, 2012. ISSN 09507051.

K. G. Shin and N. D. McKay. Robot path planning using dynamic programming. *The 23rd IEEE Conference on Decision and Control*, (December):1629–1635, 1984. doi: 10.1109/CDC.1984.272356.

S. Singh, T. Jaakkola, M.L. Littman, and C. Szepesvari. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000. ISSN 08856125. doi: 10.1023/A:1007678930559.

S. P. Singh and R. S. Sutton. Reinforcement Learning with Replacing Eligibility Traces. *Machine learning*, 22:123–158, 1996.

M Strube, R Sanner, and E. M. Atkins. Dynamic flight guidance recalibration after actuator failure. In *1st AIAA Intelligent Systems Technical Conference*, pages 1–16, 2004.

R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*. The MIT Press, 1998.

R.S. Sutton. Learning to Predict the Methods of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

C. Szepesvari. Algorithms for Reinforcement Learning, 2009. Draft of the lectures published in the Syntehis Lectures on Artificial Intelligence and Machine Learning Series by Morgan and Claypool Publishers.

H. van Seijen, H. van Hasselt, S. Whiteson, and M. Wiering. A theoretical and empirical analysis of expected sarsa. *2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, ADPRL 2009 - Proceedings*, pages 177–184, 2009. doi: 10.1109/ADPRL.2009.4927542.

H. H. Viet, P. H. Kyaw, and T. Chung. Simulation-Based Evaluations of Reinforcement Learning Algorithms for Autonomous Mobile Robot Path Planning. *Lecture Notes in Electrical Engineering*, 107:467–476, 2011.

N. X. Vinh. *Flight Mechanics of High Performance Aircraft*. Cambridge University Press, 1993.

H.G. Visser. Discrete Dynamic Programming, 2015. This example is taken from the lecture slides of lecture 8 used for a mandatory course, Aircraft Performance Optimization Lecture for the Master Air Transport Operation at the TU Delft.

C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.

M. Wiering. *Explorations in Efficient Reinforcement Learning*. PhD thesis, University of Amsterdam (UVA), 1999.

H. Wu and F. Mora-Camino. Knowlegde-based trajectory control for engine-out aircraft. *32nd Digital Avionics Systems Conference*, pages 1–12, 2013.

H. Wu, H. Bouadi, L. Zhong, and F. Mora-Camino. Dynamic programming for trajectory optimization of engine-out transportation aircraft. *2012 24th Chinese Control and Decision Conference (CCDC)*, pages 98–103, 2012.