Bachelor Thesis

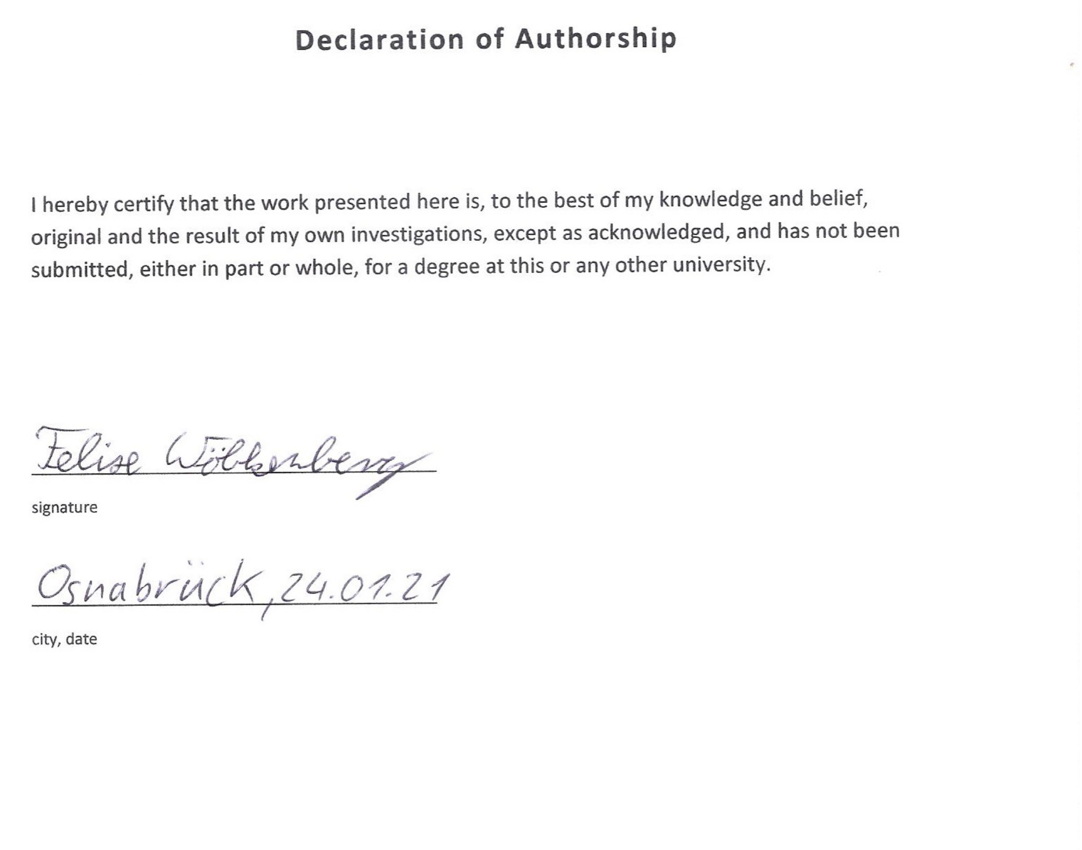
A Comparison of Nature-Inspired Algorithms for Evaluation in the Context of Computational Storytelling

Author: Felix Wöbkenberg

Supervisors:
Leonid Berov
Prof. Dr. phil. Kai-Uwe Kühnberger

Submission: January 27, 2021

Bachelor of Cognitive Science
University of Osnabrück



Declaration of Authorship

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

Felise Wöbkenberg

signature

Osnabrück, 24.01.21

city, date

Acknowledgements

First, I want to thank my supervisor Leonid Berov for the time and effort he put into supervising this thesis.

Special thanks go to Julia Wipperman, who provided the storyworld based on Robinson Crusoe.

Finally, I want to thank Thomas Otte, Adrian Schröder and Christina Wöbkenberg for proof-reading of my thesis.

Contents

List of Figures

List of Tables

Abstract

With the advancement of computational capabilities, storytelling systems became more sophisticated. In this thesis, nature-inspired algorithms were applied to generate plots in the InBloom storytelling framework. The task of scheduling happenings and finding parameters for the definition of agents was treated as an optimization problem. Different algorithms with various setups for their specific parameters were applied and compared in terms of their overall performance, using the tellability function implemented in InBloom as an objective measure to assess the quality of generated plots.

Keywords: Computational Storytelling, Happenings, Nature-Inspired Algorithms, Genetic Algorithm, Particle Swarm Optimization, Gravity Search Algorithm, Quantum Swarm Optimization

1. Introduction

With the advance of artificial intelligence, researchers strive to develop algorithms in various disciplines, that achieve and potentially surpass human-level performance. From engines playing games (Anton 2018) to computers piloting aircrafts (Kashyap 2019) to medical robots utilized in complex medical operations (Hashimoto et al. 2019), artificial intelligence algorithms are present in our everyday lives to assist and perhaps even supercede humans.

A research domain that has gained popularity in the past few decades is computational creativity. Although there does not exist a definition of what creativity actually is, researchers have proposed numerous approaches to model the concept of creativity with computational approaches (Boden 1990, Wiggins 2006).

The question whether machines or programs can be creative is anything but new. One of the first persons reported to elaborate on this topic was Ada Lovelace (Bowden 1953). Ada Lovelace and Charles Babbage worked together on a hypothetical machine called the "Analytical Engine" (Green 2005). Even though the machine was never built, it was a holistic design of a mechanical general-purpose computer.

A specific field in the domain of computational creativity is computational storytelling. Storytelling is a particularly interesting creative task, because stories are encountered on a daily basis. There have been a plethora of systems, capable of generating stories in various forms ("Robot With Mechanical Brain Thinks up Story Plots" 1931, Meehan 1977, Ryan 2017). With the increasing computational capabilities, storytelling systems became more complex over time. There is currently no definition for what makes a story (Gervás 2009) but there are approaches for determining, how suited a set of events is to be told (Labov 1972).

In this thesis, the InBloom[1] storytelling system will be used. It is based on on the multi-agent simulation system Jason by Bordini and Hübner (2007), which simulates the behavior of agents in a predefined environment, based on the Belief-Desire-Intention framework by Rao and Georgeff (2001).

In InBloom, agents have a personality based on the Big-Five model by McCrae and John (1992). Their behavior is impacted by their associated personality in order to make more believable characters (Berov and Kühnberger 2018).

[1]https://github.com/cartisan/inBloom

Artificial intelligence algorithms have been used for the automation of creative processes multiple times ((Dartnall 1994), (McCormack and d'Inverno 2012)). Since information about the environment and possible courses of actions for agents is perfectly described within the storyworld, in this thesis the problem of storygeneration will be assessed as an optimization problem. In order to generate stories and optimize their respective tellability, nature-inspired algorithms are applied.

In Chapter 2, I will explain the theoretical background for computational storytelling and nature-inspired algorithms. Information about the architecture and details about the storyworld can be found in Chapter 3.1. How the problem of generating stories was specified as an optimization problem will be described in chapter 3.2. The actual implementation of the algorithms and their respective performance will be presented in Chapter 4. A comparision between the algorithm's performance will be drawn in Chapter 5.1. Further applications of nature-inspired algorithms in the domain of computational storytelling will be presented in Chapter 5.2.

2. Background

2.1. Computational Storytelling

Storytelling is an important part of human communication and plays a central role in our everyday lifes. Stories are used to share information about past experiences, present events and can be used to convey thoughts about the future or provide entertainment in movies, literacy and games. Therefore, it comes to no surprise that storygeneration is a popular field in the domain of research (Valls-Vargas, Zhu, and Ontanon 2017).

The first artificial storygenerators predate the existence of the processor. In the article “Robot With Mechanical Brain Thinks up Story Plots” (1931) Hill presented a mechanical robot that could generate stories in form of plot lines.

With the help of modern computers, computational storytelling system became more advanced eventually. Ryan (2017) describes a fairy-tale generation system developed by Grimes in 1960, that is amongst the first to utilize artificial intelligence. It used a grammar-based approach in order to produce complete stories in the form of text. The currently only remaining known output of the system appeared in “Exploring the fascinating world of language” (1963). ”The intelligent use of referring expressions and of the discourse marker ‘thus’ position this system [was] at the cutting edge of natural language generation in the early 1960s.” (Ryan 2017, p.6)

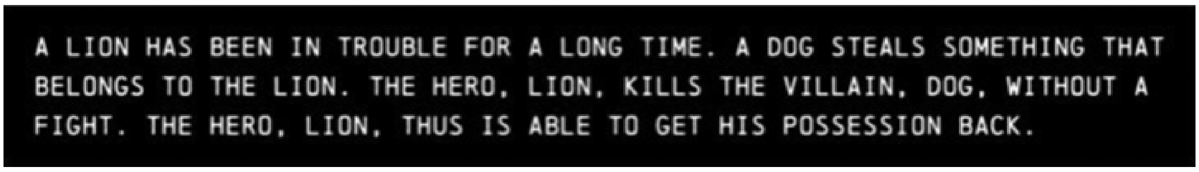

```
A LION HAS BEEN IN TROUBLE FOR A LONG TIME. A DOG STEALS SOMETHING THAT
BELONGS TO THE LION. THE HERO, LION, KILLS THE VILLAIN, DOG, WITHOUT A
FIGHT. THE HERO, LION, THUS IS ABLE TO GET HIS POSSESSION BACK.
```

Figure 1: Story by Grimes fairy-tail generation system (“Exploring the fascinating world of language” 1963)

One well-known story generation systems is TALE-SPIN by Meehan (1977). It was the result of flipping around a system, that was originally designed to understand the narrative of a story making use of conceptual dependency theory, to generate stories instead. TALE-SPIN introduced character goals as triggers for actions. Multiple characters were realized by introducing separate goal lists for each of them. Although modern story generators are more complex, most systems utilize the same approach of character- and goal-directed behavior (Langohr 2011).

According to Gervás (2009), there is currently no definition for what makes up a story. However, he provides a distinction between two levels for the conceptual representation of a story, namely fabula and discourse (Gervás (2009), p.3). Fabula is concerned with what is told in the narrative while discourse is defined as the way the story is presented.

In the aforementioned algorithms, Hills mechanical robot generated fabula in form of a plot, which consists of a chronological ordering and causality between events (Foster 1927), while Grimes fairy-tail generation system and TALE-SPIN were capable of full text representation.

Regardless of discource, a central question in computational storytelling is how to determine the quality of generated fabula. In Labov (1972) tellability was introduced as a measurement of how suited a specific set of events is to be told. Ryan (1991) expanded on this concept, using tellabilty to assess the plots quality by analyzing intrinsically aesthetic structures.

While Ryan identifies four aesthetic principles, Berov (2017b) also considers the dynamic aspect (Ryan 1991, p.153) to be one of them. Therefore, Berov (2017b) defined the five principles as stated in the table below:

1. Semantic Opposition	“’reversals in the fortunes of characters’ and contrasts between ’goals of characters with the results of their action’s” (Berov 2017b)
2. Semantic Symmetry	“structural similarities in sequences of events pertaining to different characters, or to the same character at different stages of the plot” (Berov 2017b)
3. Functional polyvalence	“the same event fulfilling several narrative functions at the same time” (Berov 2017b)
4. Suspense	“a delay between the adoption of a goal by a character and its achievement/ failure” (Berov 2017b)
5. Dynamic points	“the violation of a character’s expectation” (Berov 2017b)

Table 1: Aesthetic Principles of Tellability by Berov (2017b)

In InBloom the first four of these principles are currently implemented. Additionally, the assessment of the principles is balanced in order to avoid repetitive structures in the plot. A detailed description of the tellability function in InBloom can be found in Chapter 3.1.2.

2.2. Nature-Inspired Algorithms

There are certain real-world applications for the formulation of computational tasks as optimization problems. At the heart of every optimization problem, solutions, that either maximize performance or minimize costs in a problem-specific domain, are looked out for. Often both of these aspects must be balanced in order to achieve finding solutions of decent quality. To find these solutions can prove to be quite difficult. Search spaces of such problems may be discontinuous and heuristics may be vague such that there can be no clear answer on how to modify solutions in order to improve the overall outcome.

Artificial intelligence researchers have invented a multitude of different algorithms utilizing principles found in nature, which are used to overcome computational restrictions classical algorithms might have. According to Fister Jr. et al. (2013), most of such algorithms are inspired by successful biological systems, but there are also many successful algorithms drawing inspiration from domains like physics, chemistry or non-nature based domains like social-behaviour studies.

Different nature-inspired algorithms exhibit differences in performance on the same problem, and make different tradeoffs between local and global search. Both of which are qualities needed to find solutions of high quality in unpredictable search spaces.

To study the properties of different nature-inspired algorithms, and thus enable researchers to select the most suitable algorithm for their problem, extensive studies were conducted on well-understood benchmark problems (Stützle and Dorigo (1999), Djannaty and Doostdar (2008) and Bansal and Deep (2012)).

There are two prerequisites needed in order to apply a nature-inspired algorithm to a problem.

First, a formal representation for candidate solutions, referred to as individual, must be found which encodes the information needed to solve the problem. A typical method is to use a binary string or a string of real numbers with a fixed length. Secondly, an objective function is needed, which interprets the information encoded in such a candidate solution and returns a measurement of the solutions quality. The solutions quality is commonly referred to as fitness of an individual.

If these criteria are met, modification rules in order to improve existing solutions can be formulated. The procedure is summarized by Yang and Karamanoglu (2013) in the following generic equation:

$$[x_1, x_2, \ldots, x_n]^{t+1} = A([x_1, x_2, \ldots, x_n]^t; (p_1, p_2, \ldots, p_k); (w_1, w_2, \ldots, w_m))$$

Where:

$[x_1, x_2, \ldots, x_n]^t$	Is the population of n solutions at iteration t and its set of solutions $x_i, (i = 1, 2, \ldots, n)$.
$(p_1, p_2, \ldots, p_k)$	Is a set of k algorithm-dependent parameters
$(w_1, w_2, \ldots, w_m)$	Is a set of m random variables

Therefore, a nature-inspired algorithm A modifies a set of current solutions present at iteration t according to its transformative rules in order to generate a new set of solutions for the next iteration $t + 1$.

Yang (2020) divided nature-inspired algorithms into two categories depending on their strategies to find new solutions: procedure-based and equation-based algorithms.

2.2.1. Procedure-Based Algorithms

Algorithms of this category select solutions from the exising population and apply modifications to a single or multiple bits of information. A prevalent example for this category is the GA approach. It has many different applications but is primarily used for optimization problems. An overview of the algorithm is provided in form of pseudocode as provided by Chrominski, Tkacz, and Boryczka (2020). I will give a brief explanation of the pseudocode in this section, while a detailed description of its properties and my actual implementation will be presented in Chapter 4.2.

Algorithm 1: Genetic Algorithm

```
Data: population of individuals;
Result: Best individuals;
begin
    Initialize population;
    Evaluate individuals;
    while stopping criterion is not met do
        Select individuals for modification from the population;
        perform crossover operation;
        perform mutation operation;
        evaluate individuals;
        replace old population;
    end
end
```

First, an initial population consisting of n individuals is generated and their corresponding fitness is determined. When creating a population, it may be practical to identify problem-specific heuristics in order to improve the quality of the solutions at iteration 0, if such heuristics are available.

Subsequently, in every iteration until a termination criterion is met, individuals will be chosen to get modified by the genetic operators, namely crossover and mutation. Mutation operators apply modifications to a single individual while crossover operators exchange information between two or more individuals, so called parents.

Whenever a candidate solution is found, its performance is determined by the objective function.

The population for the next iteration $t + 1$ will consist of the best individuals from the old population combined with the newly generated solutions.

The exact definitions of genetic operators especially for mutation can vary and should be optimized to suit the problem in order to increase the quality of modified solutions. Also different selection and replacement strategies can be formulated in order to improve the performance of the algorithm.

According to Yang (2020), other notable algorithms in this category include evolutionary strategy (Aldous and Vazirani 1994), which works similar to the genetic algorithm, but only uses the mutation operation, and ant colony optimization (ACO) (Dorigo, Birattari, and Stutzle 2006). In the latter algorithm modifications to solutions are not realized via mutation and crossover, but rather the simulated deposition and evaporation of an artificial pheromone.

2.2.2. Equation-Based Algorithms

While solutions in procedure-based algorithms might be formalized in many different ways, many equation-based algorithms use a rather similar design: For a D-dimensional search space the solutions are defined as vectors x_i of length D.

Additionally, algorithms of this category also use the same scheme for solution modification, where a new solution x_i^{t+1} is found by combining vector x_i^t with a modification increment Δx_i^t.

$$x_i^{t+1} = x_i^t + \Delta x_i^t$$

While there is a plethora of different nature-inspired algorithms falling into this category, different formulas and methods for finding Δx_i^t are the main differences between equation-based algorithms.

As done for Procedure-Based Algorithms I will provide pseudocode for one popular algorithm of this category and briefly discuss it here. The pseudocode is a modification of the code provided by Tian and Shi (2018) More details about its properties, the rationale for choosing this algorithm and the practical implementation can be found in Chapter 4.3.

Particle swarm optimization (PSO) uses mathematical principles derived from swarm intelligence in order to traverse the search space of an optimization problem. Solutions are imagined to be positions of particles moving through a D-dimensional hyperplane. A particle is defined by its position in the search space x_i^t at iteration t, the particle's best solution found so far x_i^* and its current velocity v_i^t.

The velocity information is used as a vector for the modification increment Δx_i^t.

Algorithm 2: Particle Swarm Optimization Algorithm

```
Data: population of individuals;
Result: Best individuals;
begin
    Initialize population;
    while stopping criterion is not met do
        Evaluate fitness of the particle swarm;
        Find global best $g^*$ particle;
        for n=1 to number of particles do
            Update the velocity of particles;
            Update the position of particles;
        end
    end
end
```

In the first step an initial population is generated.

In every iteration of the algorithm the current fitness of the particles will be evaluated.

Afterwards, the global best g^* and the personal best x_i^* for every particle will be determined. These will be used in the velocity update function.

In the next step for every particle the velocity vector v_i^t will be updated as shown in the formula below.

At the end of each iteration all particles move through the search space according to their corresponding velocity.

$$\begin{aligned} x_i^{t+1} &= x_i^t + v_i^{t+1} \\ v_i^{t+1} &= v_i^t + \Delta v_i^t \\ \Delta v_i^t &= \alpha\varepsilon_1[g^* - x_i^t] + \beta\varepsilon_2[x_i^* - x_i^t] \end{aligned}$$

Where g^* is the global best solution and α and β are two numbers on the interval of [0,1] which define the particles strength of attraction to the global best g^* and its personal best solution x_i^* found.

ε_1 and ε_2 are two random numbers from the interval [0,1]. They are used to introduce randomness into the equation which can be benefitial for better search space exploration as demonstrated in Trelea (2002).

Therefore every particle will be attracted towards the linear combination of the global best and its personal best position found.

3. Methods

3.1. Architecture

3.1.1. Overall Framework

In this thesis, The InBloom storytelling system created by Leonid Berov is used, where affective reasoning is used to model the plot of a story in a multi-agent simulation system (Berov and Kühnberger 2018).

InBloom is based on the multi-agent framework Jason as provided by Bordini and Hübner (2007). Jason interpretes an extended version of AgentSpeak, which was originally introduced by Rao (1996). The system models cognitive agents who act and react based on their believes, desires, intentions (Rao and Georgeff 2001).

In InBloom agents are defined by their personality, which is based on the Big Five model defined by McCrae and John (1992). An agents personality impacts its behavior, in order for the character to be more believable (Berov and Kühnberger 2018). Environmental Stimuli trigger emotional reactions in the agent, which are reflected in form of an internal state called mood. The mood of an agent is represented in form of a position in a three-dimensional space based on the PAD temperament model (*Pleasure*, *Arousal* & *Dominance*) proposed by Mehrabian (1996). Personality majorly influences mood by defining the default emotional state of an agent.

The system generates stories based on simulations of agents behavior in a predefined environment. The set of possible actions for an agent to choose from, is specified in the model of the storyworld. Aditionally events, so called happenings, can take place at specific time points or conditions and may alter the environment or affect a target agent's mental state.

The output comes in form of plain text describing the simulations state during every step and as a plot graph. The graph contains all actions and events that have taken place and also includes information about causal relations and the agents internal state.

Classes provided by InBloom are mostly abstract. For a minimal implementation domain-specific subclasses for the following classes must be implemented:

- Agent
- Plot Model
- Plot Environment
- Plot Launcher
- Happening

In this thesis, an environment based on the story of Robinson Crusoe is used as descibed in Chapter 3.1.3 . The original version was provided by Wippermann (2020).

An overview of the InBloom system in form of a UML class diagram can be found on GitHub [2]. A shortened version is displayed in Figure 2 on the next page.

In order to better visualize the differences and similarities between the two realizations for tellability optimization in the Robinson Crusoe domain, I designed the diagram to look similar to the UML class diagram provided by Wippermann (2020) which is also accessible via GitHub [3].

[2]https://github.com/FelixW/BachelorThesisMaterials
[3]https://github.com/Juwipp/ThesisMaterial_UML

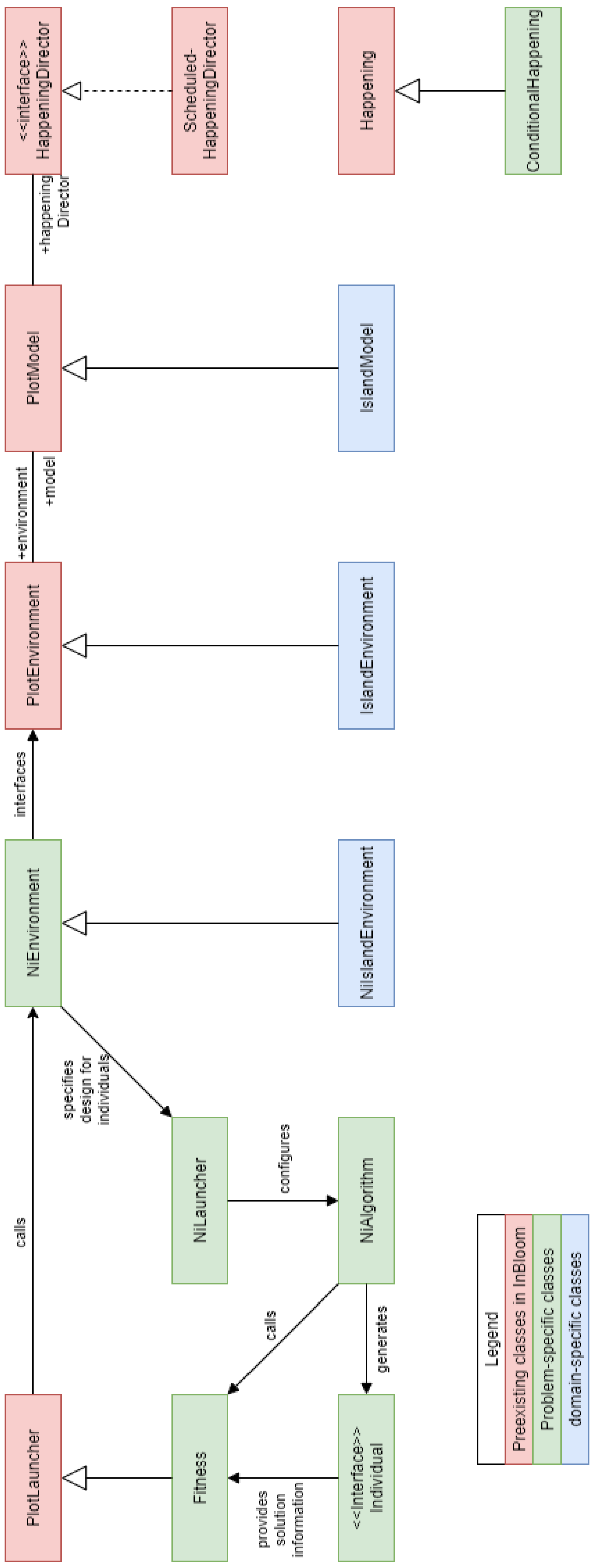


Figure 2: Shortened InBloom UML Diagram

InBloom Classes

In this section, the InBloom classes presented in the UML diagram will be explained. All classes listed in this section are of abstract nature and need domain-specific extensions.

A description of the domain-specific subclasses for *PlotModel* and *Happening* for the Robinson environment can be found in Chapter 3.1.3.

Agent The *Agent* is specified in the AgentSpeak Language as explained by Bordini and Hübner (2007, p.7). For a minimal implementation, an *Agent* must have a set of beliefs and a set plans, which prescribe the behavior *Agents* behavior in a certain environment.

During the simulation the agents actions and its perceptions of events in the environment form new and modify existing beliefs. The *Agent* chooses a single intention (= particular courses of actions) per simulation step according to his mental state. A detailed description of the agent architecture can be found in Bordini and Hübner (2007, p.8).

In InBloom, an *Agents* plans also depend on the characters personality and current mood. In the example of the Robinson environment, the *Agent* will only go on a cruise if the personality parameter ”openness” exceeds a certain range.

The InBloom system initializes an *Agent* by encapsuling the specific parameters in the class *LauncherAgent*. This class is then used to generate the required files to start the Jason multi agent system.

Happening *Happenings* define external events that can take place and apply changes to the environment or the agent. A *Happening* is defined by conditionals and its implications onto the environment or affected agents.

The conditionals are defined by the trigger determining the simulation step the happening is to be executed and its causal property which denotes presuppositions that must be met in order for the happening to have an effect. Implications of a happening are the patient (= agent being affected), the effect it has on the patient or environment and the effect it has on the mental state of perceiving agents.

Happenings extending the subclass *Conditional Happening* only occur if the causal property is met rather than taking place without an effect.

In this implementation the trigger specifies the earliest step a happening can take place rather than the exact step as explained in Chapter 3.2.1.

HappeningDirector The interface *HappeningDirector* defines methods for the plot model to access happenings during a simulation. The preexisting *ScheduledHappeningDirector* is an actual implementation of *HappeningDirector* that returns a list of*Happenings* that are *scheduled* to take place at a certain simulation step. If the conditionals allow for it a *Happening* can stay in the list of options for multiple simulation steps. It will only be removed after it actually took place.

PlotModel All the information about a storyworld is contained and managed in the domain-specific *PlotModel.* The locations of the storyworld and their associated current state, the agents actions and their specific preconditions and effects, items and other entities are described in this class.

PlotEnvironment The *PlotEnvironment* manages the communication between different classes in InBloom. It relays the action requests from the *Agent* to the *PlotModel* and in turn passes percepts from the storyworld back to the *Agent.* Each piece of information communicated will be transmitted to the *PlotGraphController* in order to be represented in the plot graph. *PlotEnvironment* is also responsible for the java initialization of the respective agents and happenings and handling termination criteria of a simulation.

The Jason Architecture calls the function *executeAction()* which informs the system about the effects on the environment of an agents action.

PlotLauncher The *PlotLauncher* is used for running a simulation of a storyworld. It is responsible for the Jason multi-agent system setup and execution. After a simulation *PlotLauncher* uses *GraphAnalyzer* to determine the tellability of a plot as described in Chapter 3.1.2.

Nature-inspired Algorithm Classes

I will give a brief description of the problem-specific classes concerned with nature-inspired algorithms. Actual implementations of *NiAlgorithm* and *Individual* can be found in Chapter 4.

Individual The interface *Individual* specifies a number of methods, candidate solutions have to implement in order to make their contained information accessible.

A description of the contents of candidate solutions can be found in Chapter 3.2.1.

NiEnvironment The abstract class *NiEnvironment* manages the communication between the nature-inspired algorithm classes and the InBloom system. Similar to *PlotEnvironment*, it initializes the *Agents* and *Happenings* based on the information provided by an *Individual*.

Extensions of this class specify the domain-specific design of candidate solutions. The design depends on the number of agents and happenings that can be scheduled, so called dynamic happenings, existing in the storyworld. Additionally static happenings can be hard-coded, such that they will be guaranteed to occur if the respective causal properties are met.

NiAlgorithm Extensions of the abstract class *NiAlgorithm* generate *Individuals* based on the design specified in *NiEnvironment*.

The abstract class *NiAlgorithm* provides general methods that are used across algorithms further described in Chapter 3.2.4. Furthermore serializing of the results and options for visibility of current performance during runtime are implemented.

NiLauncher The class *NiLauncher* instantiates the *NiEnvironment* and initializes the *NiAlgorithm* with its associated algorithm-specific parameters.

Fitness The *Fitness* class is an extension of *PlotLauncher*. It extracts the information from an *Individual* and hands it over to the *NiEnvironment* to initialize the *PlotModel*. Afterwards, a simulation is run in order to determine the *Individuals* quality in terms of tellability.

3.1.2. Plot Quality Estimation

Lehnert (1981) proposed a system, where Functional Units are used in order to summarize the plot of a story. Functional Units are defined as relevant points for summarizing a plot. Wilke and Berov (2018) used the subgraph-isomorphism algorithm in order to identify Functional Units in a given plot graph. In the resulting connectivity graph all Functional Units are listed and connected by vertices which represent the agents corresponding mental state.

The resulting connectivity graph can be used for the computation of tellability as demonstrated in Wilke and Berov (2018, p.8-9).

As mentioned in Chapter 2.1, four of the five principles are currently implemented in InBloom. A brief description of the computation of the principles can be found in the table below.

Semantic Opposition	Contrast between characters intentions and their respective outcomes Strong fluctuation over a short period of time in the agent's mood graph
Semantic Symmetry	Reoccurrence of a specific structure in the plot graph
Functional polyvalence	Vertex in the graph belonging to more than one Functional Unit
Suspense	Step difference between an intention and its final actualization

Table 2: Realization of Computing Tellability in InBloom

Since *Semantic Opposition* is influenced by changes in the mood graph, we can see that an agents personality not only influences the tellability score by impacting decision making but has a direct influence on tellability computation aswell.

In InBloom, the tellability function rates the quality of a plot based on the connectivity graphs conformity to the aesthetic principles. The result is a real number in the interval [0;1]. The higher the returned value, the better the plots quality is deemed.

The score is computed by counting the vertices associated with a respective principle and dividing this number by the total number of vertices (Wilke and Berov 2018). In the version of the tellability function used in this thesis, the rating for the first 3 principles (*Opposition*, *Symmetry* & *Suspense*) is balanced linearly in a triangular manner. If the rating of a certain principle is close to the extremes 0 or 1, the balanced value will be 0. Accordingly, a rating of 0.5 will be balanced to 1. Therefore, repetitive structures, i.e. a plot where the agent only pursues a single intention over an extended period of time, will be assessed with a lower value.

The overall tellability of a plot is defined as the average of the scores for the implemented principles.

3.1.3. Case Study: Robinson Crusoe

The storyworld used in this thesis is based on the novel "Robinson Crusoe" by DeFoe (1719). The model consists of a single character, namely Robinson, and defines several actions the agent can execute and happenings to occur.

The story always starts with the agent deciding whether he goes on a cruise based on his personality. If he decides to go, a static happening is scheduled to sink the ship. "Fortunately" Robinson manages to rescue himself to an island. From here on out multiple plotlines are possible where the agent either gets rescued at some point or stays on the island for the rest of the simulation.

Therefore, the algorithms have to first recognize the suited personality structure for the agent to go on the cruise. Subsequently, personality parameters need to be optimized and happenings must be scheduled, such that interesting plot structures may emerge.

The actions available to the agent and dynamic happenings of the storyworld will be explained in the following paragraphs.

Actions Actions can be divided into two different categories. The first one consists of a set of actions, which will reoccur regularly throughout the simulation. The other category defines reactions of the agent towards happenings.

There is a priority system associated with plans for actions. Priorities determine which plan the agent prefers to execute, when there are multiple competing options to choose from. Actions associated with survival, namely *findHealingPlants()*, useHealingPlants() and those described in maintenance schedule, have a higher priority than other actions. One exception is *extinguishFire()* which has the highest priority to be executed.

Maintenance schedule There are two cyclic courses of action defined in the environment, which must be executed successfully in order for the agent to survive. Both cycles have a trigger to initiate and an action finishing the sequence. In order to complete the final action, certain criteria must be met. The execution of the final action resets the cycle to start anew.

The sleep cycle has a maximum length of 24 simulation steps. At two thirds of the cycle the agent will get tired, resulting in the initiation of the plan sleep. In order to accomplish the goal to sleep, Robinson must first seek shelter by building a hut.

The other cycle is concerned with robinsons diet. Its duration amounts to 12 steps and the trigger of getting hungry is perceived after the first half of the cycle has passed. In order to eat, the agent must have food in his inventory. He first has to successfully find food by executing the action *getFood()* to subsequently perform *eat()*.

Therefore, the agent is occupied during several steps of the simulation, such that happenings can be scheduled to interfere with these reoccurring plot lines. If the agent fails to complete one of the cycles during its respective length, it will lead to the agents death.

The length of the cycles has been chosen to be a reference to real time. Since the InBloom storytelling system has no concept for time spent, this is meant to be a first step towards an actual day and night cycle. The triggers of the respective cycles occur after different fractions of their cycle length have passed in order to avoid the cycles clashing naturally.

Reactions to Events Reactive actions are defined by a precondition and the corresponding effect on the agent or environment. In contrast to the previous category these actions do not occur by default but are executed when the respective requirements are met, which is realized by happenings taking place.

Action	**Precondition**	**Effect**
findHealingPlants	Agent is poisoned, Agent has no healing plants in his inventory	Agent adds healing plants to his inventory
useHealingPlants	Agent is poisoned, Agent has healing plants in his inventory	Agent uses plants to cure poison status
extinguishFire	Island is burning	Island stops burning
findFriend	Agent feels homesick, Agent has no friend	Agent has a new friend
complain	Agent feels homesick, Agent has a friend	Agent stops feeling homesick
goOnShip	Agent is on the island	Agent gets rescued from island

Table 3: Reactive Actions in the Robinson Storyworld

Happenings In the Robinson storyworld there are a total of 9 dynamic happenings. Their varying effects on the agent and the environment are described on the following page.

Happenings impacting maintenance Happenings in this category interfere with the general maintenance cycle of the agent. By delaying the agents goals, the *Suspense* and *Opposition* scores of the tellability function are increased. If the cycle is delayed for too long, the agent might die and the simulation will be terminated.

Happening	Condition	Effect
FoodStolen	Agent has food in his inventory	Agent has no food in his inventory anymore
FoodPoisoning	Agent has food in his inventory	Agents food inflicts poison status if eaten
Storm	There exists a hut on the island	Hut gets destroyed

Disruptive happenings These happenings prevent the agent from executing actions in various ways. Similar to the previous category, they lead to an increase in *Suspense*, *Opposition* and might even lead to the agents death.

Happening	Condition	Effect
Fire	Island is not burning	Island is burning
Rain	It is not raining	It starts raining
PlantDisease	Island has healing plants	Island does not have healing Plants anymore

Social happenings Happenings in this category interact with the social actions *findFriend()* and *complain()*. Since multiple functional units are involved in social actions, they increase the polyvalence rating of the plot.

Happening	Condition	Effect
Homesick	No Condition	The agent feels homsick
LoseFriend	The agent has a friend	The friend dies

Ending simulation The *ShipRescueHappening* is unique in its effect to end the simulation. It cannot occur before simulation step 5 (see: Chapter 5.2.2).

Happening	Condition	Effect
ShipRescue	Minimal plot length	The Agent gets rescued from the island.

3.2. Formulation as Optimization Problem

3.2.1. Problem Encoding

The algorithms presented in Chapter 4 interface with the InBloom system via the definition of an agents personality and the scheduling of happenings. As mentioned in Chapter 3.1.1, agents are defined by 5 personality parameters. These parameters are specified to be a real number $\in [-1; 1]$. Since the causal properties of happenings are defined in the problem-specific extension of *NiEnvironment*, the algorithms only need to find a trigger in form of an integer and a patient to be affected by the happening.

In the Robinson storyworld, all happenings are extensions of the *ConditionalHappenings* class. Hence happenings will only take place, when their causal properties are met. Therefore the triggers are defined as the earliest simulation step a happening can take place rather than the exact simulation step. Instead of determining the exact point in time a happening can take place - which is an impossible task because we do not know the state a simulation will be in for any time step - we only have to find a step previous to the earliest step the causal properties are met. To effectively schedule a happening is now a trivial task: By setting the trigger to simulation step to "1", a happening will be guaranteed to take place, provided there is a situation where the causal properties are fulfilled.

If there are multiple instances where a happening would have an effect, the value for the trigger denotes the next closest point in time for the happening to occur.

I will make use of two 2-dimensional vectors per candidate solution. Although the Robinson Crusoe storyworld features only one agent, the solution vectors and algorithms are designed to be able to handle multiple agents. For an environment with n agents and m happenings, the vectors are defined as described below:

The first solution vector is used to define the personality of the storyworlds agents. Since we have n agents and 5 parameters per agent, it follows that the vector is of size $n * 5$ containing real numbers of the interval [-1;1].

The other vector encodes the scheduling of happenings. In the scope of this thesis, we consider exactly one occurrence of each happening per agent in the environment. Hence, the vector can be designed to be of size $n * m$ containing integer numbers. A positive number t at position i, j will schedule the happening with the index j to the i^th agent at simulation step t. Negative numbers or 0 will deactivate a happening accordingly.

The length of the simulation depends on the latest scheduled happening plus a number of extra steps for the happening to have an effect according to the following formula:

$$MaxLength = Max(happeningVector) + Buffer$$
$$Buffer = \sqrt{Max(happeningVector)} + 1$$

A simulation can end earlier in case of all agents dying or specific happenings (here: ship rescue) ending the simulation early. In order to avoid too long simulations, the maximum length per simulation is capped at 100 steps.

3.2.2. Properties of Search Space

The problem of finding the best story in a storyworld is of particular difficulty because of the associated search space. The conceptual space of all possible stories in the Robinson Crusoe storyworld is infinitely large since values in the solution vectors are not discretized. Even if the personality parameters would be discretized to meaningful intervals of 0.05, combined with the hard cap of 100 simulation steps, there would be

$$41^5 * 101^9 = 1.26 * E^{26}$$

126 septillion possible permutations.

Due to the solution vector design and the framework in use, the search space is highly discontinuous. For example, as described in the previous section, small changes to the happening trigger may exceed certain thresholds, resulting in the happening not getting activated and therefore altering the plot drastically, or hanot having an effect at all.

Additionally, there is no clear cut heuristic, according to which candidate solutions could be improved. Since there is no way to know what actually happens during a simulation, it cannot be determined, whether a specific parameter has an effect on the overall tellability.

3.2.3. Aggravating Circumstances

There are some tradeoffs for using the Jason framework. First of all, the system operates very slowly. The time for running a single simulation usually varies between 1 and 30 seconds. Accordingly, the variance in number of candidate solutions an algorithm can look at during a specific period of time, is extremely high. Hence, we cannot draw meaningful conclusions about an algorithm's performance in regard to the amount of analyzed solutions.

In Jason, agents will go through multiple reasoning cycles in every simulation step. In the InBloom framework the amount of time a simulation step may take is limited to a certain amount. Therefore, if an agent does not manage to finish his reasoning process in the specified time window, his actions will be delayed. As a result, there is a small chance during every simulation step, for the plot graph to be altered. In Figure 3, there are two different plots from the same candidate solution. The correct result is displayed on the left side, while the action of going on a cruise has been delayed in the plot on the right side.

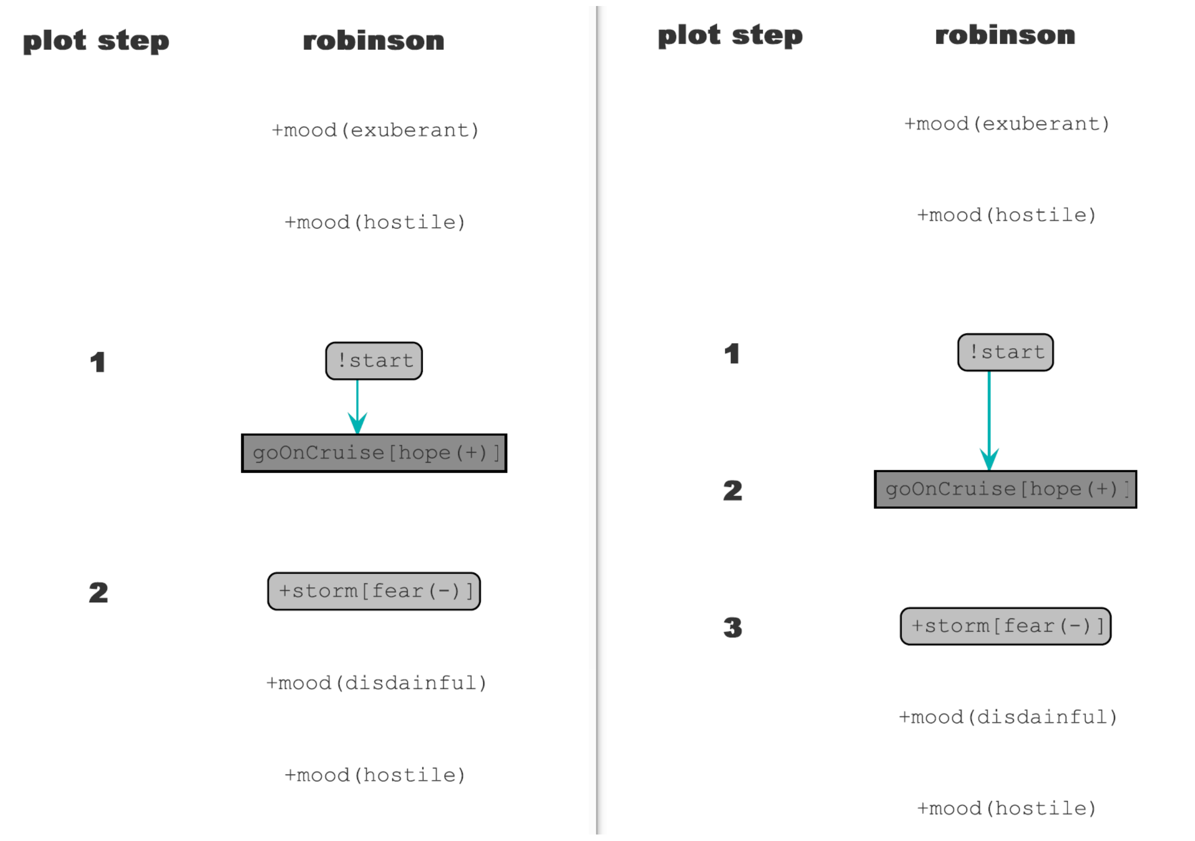


Figure 3: Example for Inconsistencies in Jason

The tellability of the plot displayed on the right side of Figure 3 is around 10% higher than the tellability of the correct version. In Chapter 5.1 I will further discuss the impact of such an inconsistency on the performance of the nature-inspired algorithms. Furthermore, since the processing power of different computers is different, the results of the system may vary depending on the hardware used.

3.2.4. Procedure

In order to compare the performance of the nature-inspired algorithms presented in Chapter 4, I will perform a small statistical analysis for the various configurations. There will be 10 trials per configuration with a maximum runtime of one hour per trial. The results can be found on GitHub [4].

Since the tellability of a particular candidate solution may vary because of inconsistencies of the Jason system (see: Chapter 3.2.3), I will use the values the algorithms actually return for the statistical analysis. Due to the nature of the algorithms preferring to keep the better solutions, the best value an algorithm has found for a specific solution vector is recognized.

In the scope of this thesis, similar Variances and normal distribution in the data is assumed.

[4]https://github.com/FelixW/BachelorThesisMaterials/tree/main/Results

4. Realization

In this Chapter, the actual implementations for the algorithms will be presented. I will describe the algorithms respectively and show their specific performance. Afterwards the modifications applied to the algorithms and their impact will be discussed. The results between different algorithms will be analyzed in Chapter 5.1. A table containing the results for all algorithms can be found in the appendix (Table 18).

As mentioned in Chapter 3.2.1, the algorithms are designed to be domain-general. Therefore, some of the operators are designed specifically to work in environments with multiple agents. The particular operators will have an annotation, describing whether they are used in the the scope of this thesis.

4.1. Initialization

In order to make a fair comparison, all algorithms make use of the same operators for generating the initial population and all algorithms will run with a population of similar size. In the case of the Genetic Algorithm and Particle Swarm Optimization, a size of 20 individuals have been chosen. There are 3 operators for each of the respective solution vectors. Since we have no domain specific knowledge, we mostly depend on randomness in order to initialize candidate solutions.

Personality Initialization Personality parameters are set to be in the interval of [-1;1]. Therefore, it is not necessary to introduce additional parameters for the initialization.

Random Personality Initialization The *RandomPersonalityInitializer* simply generates random real numbers $\in [-1; 1]$ which are inserted into the solution vector.

Discrete Personality Initialization The *DiscretePersonalityInitializer* chooses numbers randomly from a set of possible values:

$$[-1, -0.9, -0.75, -0.5, -0.25, -0.1, 0, 0.1, 0.25, 0.5, 0.75, 0.9, 1]$$

The set contains values $\in [-1; 1]$ from an interval of 0.25. There are additional values close to the extremes (0.9 & -0.9) and the middle (0.1 & -0.1).

Steady-Discrete Personality Initialization This operator works exactly as the previous initializer but chooses only unique values per agent.

Happening Initialization Unlike personality parameters, happenings do not have a fixed interval of permissible values. Hence, an integer number *Maximum_Length* is used as the maximum step a happening can be scheduled during initialization. During the algorithms runtime, the length of a simulation will alter based on its performance and the operators applied.

Random Happening Initialization Similar to the previous random initializer, the *RandomHappeningInitializer* generates random numbers in $\in [0; \textit{Maximum\_Length}]$ for the candidate solution.

Probabilistic Happening Initialization The *ProbabilisticHappeningInitializer* schedules happenings with the probability of $1/Number\_Agents$. The steps the happenings are scheduled to are multiples of *Maximum_Length* divided by the total number of happenings in the environment. As a result, happenings are either scheduled to occur at the same time or there is a certain minimum temporal difference in the scheduling.

Steady Happening Initialization This operator is designed for environments with multiple agents, therefore it is not used throughout this thesis. The *SteadyHappeningInitializer* works similar to the previous initializer, but schedules every happening exactly once per agent.

Setup For every combination of the operators and two different values for *Maximum_Length*, 100 individuals have been generated. As values for *Maximum_Length* 30 and 50 steps have been chosen. The results are displayed in Table 4 and Table 5. In every field of the tables, information about the best solution with its respective length and the average quality of the generated population can be found.

Initialization with max. Length = 30	Random PersInit	Discrete PersInit	Steady Discrete PersInit
Random HapInit	Best: 0.18 with Length: 28 Average: 0.0302	Best: 0.15 with Length: 33 Average: 0.032	Best: 0.186 with Length: 24 Average: 0.037
prob. HapInit	Best: 0.174 with Length: 24 Average: 0.029	Best: 0.17 with Length: 28 Average: 0.41	Best: 0.145 with Length: 24 Average: 0.032

Table 4: Initialization Quality with max. Length = 30

Initialization with max. Length = 50	Random PersInit	Discrete PersInit	Steady Discrete PersInit
Random HapInit	Best: 0.155 with Length: 42 Average: 0.035	Best: 0.171 with Length: 38 Average: 0.036	Best: 0.172 with Length: 38 Average: 0.034
prob. HapInit	Best: 0.157 with Length: 22 Average: 0.037	Best: 0.158 with Length: 43 Average: 0.04	Best: 0.188 with Length: 20 Average: 0.036

Table 5: Initialization Quality with max. Length = 50

All generated individuals with tellability of at least 0.18 are below 30 steps. Therefore, I choose to initialize with *Maximum_Length* = 30 in the statistical analysis. According to Table 4, the *RandomPersonalityInitializer* yielded the worst population average. On that score, the operator will not be used.

4.2. Genetic Algorithm

4.2.1. Motivation

According to Reeves (1997), the Genetic Algorithm (GA) has become a popular approach for solving hard combinatorial optimization problems. Due to the algorithm utilizing different operators, that can be formulated and adjusted in order to work well in a problem-specific domain, the GA is very flexible in its applications (Reeves 1997, p.1).

In this implementation, I aim to propose operators, that combine the usage of randomness and heuristics in order to improve candidate solutions. Thus, the algorithm can be very flexible in getting from one point of the conceptual space to another. A downside to this strategy is the reliance on chance to successively improve candidates tellability score.

4.2.2. Base Algorithm

As described in Chapter 2.2.1, the Genetic Algorithm chooses a number of individuals from a population for the appliance of modifications. Subsequently the old population will be replaced with better performing individuals.

Individuals are specified by their genetic information. They contain two chromosomes which represent the domain specific solution vectors and their associated simulation length (Chapter 3.2.1).

In this thesis, 4 individuals get chosen from a population of size 20. Those individuals will first undergo a crossover operation, afterwards the offspring will get modified by mutation operators. The results of crossover and mutation modification form the total offspring which will be used for replacement of the population.

The specific genetic operators are described below:

Selection In this thesis, the classic roulette-wheel strategy is used for selection. The *RouletteWheelSelector* chooses individuals randomly, with better performing individuals having a higher chance to be selected.

Crossover Crossover operators aim to exchange allele (= value for a specific parameter) between at least two so-called parents. Although operators may work differently, they often make use of a probabilistic parameter *CrossoverProbability*. Each crossover operator constructs exactly two children, which will be handed over to the offspring.

Chromosome Crossover The operator works parameter-free. As the name suggests, the offspring created by this operator will have exactly one chromosome of each parent. *ChromosomeCrossover* has limited exploratory value but may work well in respect to convergence.

Binomial Crossover During *BinomialCrossover*, the solution vectors containing the genetic information of the parents will be iterated. Allele will be exchanged between the respective parameters with a chance of *CrossoverProbability*.

X-Point Crossover Rather than modifying single allele, the *XPointCrossover* operator exchanges strings of information between parents.

This is realized by iterating the parent's chromosomes. With a chance of *CrossoverProbability* a certain position will be denoted as a crossover point. All allele between crossover points will be exchanged. As a result, the strings exchanged will contain information of allele in the same dimension. Therefore, similar information, for example personality parameters for a specific agent or schedulings of a specific happening, will be exchanged.

Since there is no ordering in the parameters, the order in which the dimensions will be traversed is randomized.

This operator is originally intended for storyworlds with multiple agents. It is still functional for an environment with a single agent and will therefore be used in this thesis.

Voting Crossover This is the only multi-sexual crossover operator. Additionally to the initial 2 parents to be crossed over, a random number $\in [0; PopulationSize - 2]$ of individuals will be added as parents for the resultants. The first child will consist of the average value for every parameter in the parents. The other child will be given allele from random parents for every parameter respectively.

Mutation Contrary to crossover, all mutation operators work very similar in this realization. All operators iterate the respective individual's chromosomes and apply modifications with a certain chance *MutationProbability*. They mainly differ in the way new values for the allele are generated.

Random Mutator The *RandomMutator* generates random real numbers $\in [-1; 1]$ for the personality and random integer numbers $\in [0; SimulationLength]$ for happening parameters.

This modification strategy is simple but provides good explorative value.

Toggle Mutator The outcome of modifications applied by this operator depends on the chromosome type and specific value for a particular allele to be modified.

Personality parameters are multiplied by -1.

Deactivated happenings are scheduled by inserting a random integer $\in [1; SimulationLength]$.

Scheduled happenings are deactivated by setting the allele's value to 0.

Oriented Mutator For every allele $x_{i,a}^t$ that shall be modified, the *OrientedMutator* chooses a different value $x_{i,b}^t$ within the same chromosome. The allele is modified conform to the following equation:

$$x_{i,a}^{t+1} = x_{i,a}^t + \varepsilon * (x_{i,b}^t - x_{i,a}^t)$$

where ε is a random variable $\in [-1; 1]$.

Therefore the allele is either shifted towards or away from another value in the same candidate solution.

Guided Mutator The *GuidedMutator* works similar to the previous operator. Instead of looking at another value in the same chromosome, the *GuidedMutator* modifies an allele $x_{i,a}^t$ by looks at the same parameter in a different candidate solution $x_{j,a}^t$.

$$x_{i,a}^{t+1} = x_{i,a}^t + \varepsilon * (x_{j,a}^t - x_{i,a}^t)$$

where ε is a random variable $\in [-1; 1]$.

Therefore the allele is either shifted towards or away from the same value in another candidate solution.

Replacement After every iteration, the worst performing 50% of the population will be replaced with the fittest individuals from the total offspring. Thus, a constant influx of new genetic material into the population in every iteration is established.

The replacer aims to avoid including candidate solutions that already exist in the population.

4.2.3. Results

Due to limited computational resources and time restrictions, i cannot perform a complete ablation study. The results of testing different operators in isolation and combination were consistent in that the more operators are used, the better the overall outcome. Therefore, I will use all of the genetic operators explained in the previous section for the statistical analysis.

Two sets of parameters are used for the statistical analysis:

$$\mathit{CrossoverProbability} = 0.25,\ \mathit{MutationProbability} = 0.15\ \&$$
$$\mathit{CrossoverProbability} = 0.2,\ \mathit{MutationProbability} = 0.1$$

GA Base	Best	Mean Best	Variance
0.25 / 0.15	0.236	0.218	0.0001
0.2 / 0.1	0.312	0.258	0.0013

Table 6: Results of the Classic Genetic Algorithm

The set of *CrossoverProbability* = 0.2 with *MutationProbability* = 0.1 yielded very good results. The mean best is very high with 0.258 and the GA managed to find a best result of 0.312.

On a different note, the variance is also very high, showing the inconsistent performance of the GA. The algorithm can get stuck in a local optima, where significant changes must be made to individuals to further improve the tellability of individuals. Utilizing randomness seems to not always suffice to overcome a local optima in a reasonable amount of time.

In the setup of *CrossoverProbability* = 0.25 with *MutationProbability* = 0.15, the modification probabilities were most likely chosen too high for the GA to perform well. The best solution found is lower than the mean best of the other setup.

4.2.4. Floating Parameter Version

In order to overcome the inconsistent behavior of the base algorithm, I attempt to replace the static probabilities with decay functions. The intention is to track changes to the individuals that positively influence the tellability score. This positive feedback should lead to an increase in the probability for the specific parameter to get modified and for the modification operator used to get selected again.

Decay functions are well suited for the observation of changes to the parameters. They represent a low-level form of attention because they naturally implement rapid changes towards the center of their specific ranges if values are close to the extremes of the functions. Additionally, decay functions by default "forget" about old information over time. Furthermore, probabilities for specific parameters and operators never go to zero but just become increasingly unlikely. Thus, applying modifications to a certain paramete is never made impossible. Since, depending on the state of the population, different parameters might lead to increases in tellability, these are very useful properties.

The decay functions are defined for every operator and for every parameter of the solution vector. Crossover and mutation have separate sets of functions for the parameters, since they will give different feedback. For example, if the agent in the robinson environment has the correct value for the openness personality parameter to go onto the cruise, Crossover will copy that correct value onto other candidate solutions, while mutation might change the correct value.

The probability to change a specific parameter is the product of the current decay function value for the operator in use and the current decay function value of the parameter. All decay functions follow the same update rule. If a modification operation is deemed successful, the value for the decay functions of the operator and all the parameters that have been changed is increased. The criteria, whether a modification was successful, differ for operators and parameters. For operators, one of the children must have a higher tellability score than both parents. For parameters, the children must additionally achieve a higher tellability score than the population average. If a modification did not meet these criteria, values for the respective functions are decreased in the same manner.

Therefore, transformation rules may have different probabilities associated based on their performance and the state of the population. The frequency, in which a specific parameter in the solution vector is modified, depends on the feedback from recent modifications.

All decay functions follow the same update rule. A function p^t at iteration t will be updated according to the following equation:

$$p^{t+1} = \begin{cases} p^t + (\omega - p^t) * (1 - \beta^t) * \alpha & \text{if respective criteria is met} \\ p^t * (1 - \alpha * \beta^t) & \text{else} \end{cases}$$

$$\beta^{t+1} = \begin{cases} \beta^t + (1 - \beta^t) * \alpha & \text{if respective criteria is met} \\ \beta^t * (1 - \alpha) & \text{else} \end{cases}$$

Where:

ω	= Upper limit of the respective function
α	= Update rate
β^t	= Overall success rate at iteration t

The upper limit ω is different for functions representing parameters of the solution vector and those for operators . Functions for parameters will stay in the interval of [0;1], while functions for operators are capped at 0.5. All functions will be initialized to the center $\omega/2$ of their respective range.

The adaption of a function is defined by the parameter *UpdateRate* α. The *UpdateRate* defines how fast the algorithm adapts his runtime behavior.

If the population converges to a local optima, probabilities might get reduced, forcing the algorithm to terminate prematurely. Therefore the overall *SuccessRate* at iteration t β^t regulates the adaption of the decay functions. The current success rate β^t is also defined as a decay function on the interval [0;1]. If the algorithm has not managed to find improvements to solutions for extended periods of time, the decay of the functions associated with probabilities will get reduced. Positive feedback will result in higher increases accordingly.

Therefore, a function will get updated based on the update rate and the current overall success rate. We still end up having a parameter *UpdateRate* α, but it describes adaption towards a situation rather than hardcoding algorithm's behavior.

The algorithm will be run with $alpha = 0.05$ and $alpha = 0.1$ for the performance analysis.

4.2.5. Results

GA Floating	Best	Mean Best	Variance
0.1	0.319	0.2434	0.0015
0.05	0.309	0.2375	0.0016

Table 7: Results of the Floating Parameter Genetic Algorithm

The floating parameter version of the GA managed to find the total best solution with a score of 0.319 during the analysis. The overall results of the better performing setup with *UpdateRate* = 0.1 for mean best are slightly lower and variances are a little higher than the algorithm's performance in the static parameter version with *CrossoverProbability* = 0.2 and *MutationProbability* = 0.1, but the differences are not statistically significant according to the t-test with p = 0.4183.

Although the floating parameter version of the algorithm does perform well, the observation of positive changes to specific parameters does not provide a heuristic that improves the base algorithms performance.

4.3. Particle Swarm Optimization

4.3.1. Motivation

As described in Chapter 2.2.1, the Particle Swarm Optimization algorithm (PSO) (Kennedy and Russel 1995) utilizes social behavior in order to traverse the search space of a given problem. "Problem solving is a population-wide phenomenon, emerging from the individual behaviors of the particles through their interactions"(Poli, Kennedy, and Blackwell 2007).

PSO has many practical applications (Poli, Kennedy, and Blackwell 2007, p.18) and is effective for optimizing a wide range of different functions (Kennedy and Russel 1995, p.6). Additionally, the algorithm is highly customizable in order to improve its problem-specific performance (Poli, Kennedy, and Blackwell 2007, p.9).

In this thesis, the PSO algorithm has been chosen for comparison with the genetic algorithm because it brings interesting opposing properties. I start off using a deterministic version of the algorithm and apply variations subsequently.

First, in PSO all candidate solutions will be subject to a modification process and in this process all parameters will get modified. Secondly, the algorithm uses only a single update rule which, contrary to the heuristic implemented in the GA version, provides a clear-cut heuristic in how much a specific value is changed per iteration. Finally, depending on the implementation used, PSO may work completely deterministic in applying modification rules.

Typical tradeoffs for using PSO are for example its inability to quickly traverse the searchspace. Contrary to the genetic algorithm, PSO modifies candidate solutions in small steps. Therefore operations like deactivating a happening by setting its trigger to 0 are not possible. Furthermore, PSO can get stuck in a local optima. If the best performing individuals move towards a local optimum, there is no guarantee for the particles to be able to get out. In the worst case, all particles of the swarm will be attracted towards the specific local optima.

4.3.2. Base Algorithm

Candidate solutions in the PSO algorithm are called particles and contain three solution vectors. The first represents their current position x_i^t in the conceptual space. The second vector is used to remember their best position found x_i^*. The final vector contains the particle's velocity v_i^t at iteration t.

A particle moves according to its velocity through the search space. The velocity of a particle gets updated to point towards the centroid of the particle's personal best and the global best solution found.

Different implementations of the PSO algorithm often vary in their modification of velocity Δv_i^t and parameter choices for attraction forces α. In the scope of this thesis, two different update strategies combined with two parameters for attraction α are used. In the first update strategy, here referred to as Single PSO, a particle orientates in the direction of its personal best position x_i^* and the global best position. The other update strategy used is Fully-Connected PSO, where the personal bests of all particles are used for the velocity update.

Forces in PSO can be negative. While particles get attracted to better ones, better performing particles get pushed away from particles with less tellability, furthering the exploration of the search space. The realization of this process is trivial in Fully-Connected PSO. In this implementation of Single PSO, the best performing particle orientates towards its own personal best and will get pushed away from the second best particles best position. For the statistical analysis, the attraction force α will be set to 0.05 and 0.1 respectively.

In this implementation, the velocity update v_i^{t+1} has an inertia component associated. Inertia describes the particles tendency to stay at its current position in the conceptual space. Here, it is realized by multiplying the velocity of the previous step v_i^t with the inverse of the attraction $(1 - \alpha)$.

The formulas for the movement of particles can be found in Table 8 on the next page.

$$x_i^{t+1} = x_i^t + v_i^{t+1}$$

$$v_i^{t+1} = (1-\alpha)v_i^t + \Delta v_i^t$$

$$\Delta v_i^t = \frac{\alpha[x_i^* - x_i^t] + \sum_{j=1}^{n} \alpha[x_j^* - x_i^t]}{n+1}$$

Where:

α	= Force of attraction
$(1-\alpha)$	= The inertia for a particle
n	= Number of informants for a particle
	(Single PSO: n=1, Fully-Connected PSO: n = swarm size - 1)
x_i^t	= Particle's position at iteration t
v_i^t	= Particle's velocity at iteration t
Δv_i^t	= Velocity update rule

Table 8: PSO Update Rules

Another important aspect to mention is the implementation-specific behavior of the particles at the boundaries of the respective parameters spectrum. Here, personality parameters are bound to the interval [-1;1]. Accordingly, particle positions cannot exceed and will therefore stick to these boundaries. On the other hand, values for happening parameters do not have set boundaries and parameters may assume any value.

Although the modification rules of the PSO algorithm are quite simple, the resulting particle behavior may be nontrivial (Poli, Kennedy, and Blackwell 2007, p.14). This thesis is too short in order to present all the dynamics of the swarm's behavior, but to give a rough overview, some important functionalities are explained below.

A happening will be scheduled, if its trigger is set to a positive integer. Since a simulation can end earlier than the associated *SimulationLength* due to the *ShipRescueHappening* or the death of all agents, not all scheduled happenings will take place. Generally, happenings are deactivated by setting the trigger to a value <= 0 but PSO has shown to effectively use *ShipRescueHappening* in order to control the length of a simulation and therefore inserting an upper limit for happenings to occur. Reactivation of happenings depends on the particles respective position and the state of the swarm. Since a particle's movement depends on its personal best position and the performance of other particles, its velocity might get updated to point towards the activation range for happenings.

4.3.3. Results

Base PSO	Best	Mean Best	Variance	Mean Average
Single 0.1	0.281	0.227	0.0008	0.144
Single 0.05	0.281	0.218	0.0017	0.128
Full 0.1	0.3	0.238	0.0009	0.159
Full 0.05	0.272	0.214	0.0011	0.135

Table 9: Results of the Classic PSO Algorithm

Both update strategies yielded the best result with attraction force $\alpha = 0.1$. The mean best for Fully-Connected PSO is slightly higher, an unpaired t-test determines the difference is not statistically significant with a p-value of 0.4491.

There are many comparisons between deterministic and non-deterministic implementations of PSO (Peri, Diez, and Fasano 2012, Tsujimoto, Shindo, and Jin'no 2011). I did test different strategies to introduce randomness into the algorithm, but a full statistical analysis on those variations would exceed the computational limits of this thesis. Generally speaking, I did not find any performance improvements by implementing indeterministic particle behavior.

In contrast to the GA, the mean average can be interpreted in a meaningful way. It is defined as the average of all the individual particles personal best scores. Interestingly, the mean average is very low compared to the mean best, indicating that most of the particles do not reach high-scoring positions in the conceptual space.

4.3.4. Gravitational Version

In order to improve the quality of the average population, I implemented an update function based on Newtonian Gravitational Laws. There are several PSO-like systems, that make use of gravitational based functions in order to improve particle behavior, however the exact implementation of the different functions may differ (Mohd Sabri, Puteh, and Rusop 2013, pp. 9–16). The main difference to classic versions of PSO, is that the attraction force α is not defined by a static parameter but is calculated based on the particles position and performance compared to other particles.

Depending on the functions in use, lower performing particles might accelerate faster, in order to move to quicker towards interesting positions in the conceptual space. Top performing particles change their velocity more slowly, thus improving convergence behavior.

In this implementation, the particles behave according to the update rules shown in Table 8. The force $\alpha_{i,j}$ exerted on a particle x_i by another particle x_j is calculated depending on the particles inertia ι_i, the relative attraction between particles $\Delta\tau_i\tau_j$ and relative distances between particles $\Delta\rho_i\rho_j$ as described in the formula (Table 10).

$$\begin{aligned}
\alpha_{i,j} &= \iota * \Delta\tau_i\tau_j * (\Delta\rho_i\rho_j)^2 \\
\iota_i &= 1 - \tfrac{\tau_i}{\tau *} \\
\Delta\tau_i\tau_j &= \tfrac{\tau_j - \tau_i}{\tau *} \\
\Delta\rho_i\rho_j &= \textstyle\sum_x^D \tfrac{\rho_{j,x} - \rho i,x}{\rho^*_x}
\end{aligned}$$

Where:

D	= Amount of dimensions for a specific problem
$\alpha_{i,j}$	= Force of attraction
ι_i	= Inertia
τ_i	= Tellability score of a particle
τ^*	= Tellability score of the best particle
$\Delta\tau_i\tau_j$	= Difference in tellability relative to the global best τ^*
$\rho_{i,x}$	= Position of a particle i in dimension x
ρ^*_x	= Range of values for dimension x
$\Delta\rho_i\rho_j$	= Distance between particles relative to the covered conceptual space

Table 10: Gravitational PSO Force Function

In the original implementation of the Gravitational Search algorithm (GSA) (Rashedi, Nezamabadi-pour, and Saryazdi 2009), particles are attracted by all better performing particles. Thus, there are no negative forces. This principle is included in the analysis as an additional update strategy for gravitational PSO.

4.3.5. Results

Grav. PSO	Best	Mean Best	Variance	Mean Average
Single	0.269	0.211	0.0014	0.157
Gravity Search	0.24	0.195	0.001	0.139
Full	0.247	0.191	0.0009	0.138

Table 11: Results of the Gravitational PSO Algorithm

The adjustment to the particle's velocity update resulted in a worse performance compared to the classic version with $\alpha = 0.1$. However, the mean average is actually closer to the mean best than in the previous version.

There could be several reasons for the gravitational version to yield bad results. First, the fast movements of low scoring particles could lessen the exploration of the search space and we end up in a local optima early. Another reason could be that convergence is negatively impacted by the reduced update rate of well performing particles. Maybe low scoring particles have too strong of an impact on the best performing particles and therefore good solutions cannot be optimized.

The behavior of the swarm becomes too complex at this point, to be analysed by looking at the end results. Multiple variations of the gravitational function have been tested but lead to same results overall. I therefore move on to elaborate on the next algorithm.

4.4. Quantum Swarm Optimization

4.4.1. Motivation

Since replacing the static parameters with functions did not result in an increase in performance for the respective algorithms, I am going to propose a hybrid approach. The goal is to come up with an algorithm, that combines the properties of the previously presented ones.

Quantum Swarm Optimization (QSO) is a modification to the PSO algorithm, where principles from quantum theory are applied in order to improve the behavior of particles. Originally implemented by Sun, Feng, and Xu (2004), QSO algorithms make use of quantum-particles that may assume multiple different positions (=quantum superposition) in the conceptual space. There are many realizations of this approach, often distinguishing themselves in the way the state of quantum superposition is implemented. Sun, Feng, and Xu (2004) and Pant, Thangaraj, and Abraham (2008) use, conform to the uncertainty principle, wavefunctions in order to determine the likelihood of a particle being in a certain position. In Du et al. (2010) quantum particles are defined to always have exactly two positions.

In this implementation the state of quantum superposition is realized by applying the operators of the GA to the quantum-particles. Therefore, new positions for particles may appear and their behavior is modified accordingly.

4.4.2. Algorithm

The individual used in QSO is called quantum. A Quantum remembers, similar to the particle, its best position q_i^* found so far. Additionally, it may have multiple current positions in the conceptual space. For every possible current position, it holds a solution vector $q_{i,p}^t$ for the respective coordinates in the conceptual space and the associated current velocity $v_{i,p}^t$.

If the Quantum has only one current position, it updates its velocity exactly the same as a particle described in Chapter 4.3. If the quantum has multiple positions, sub-swarm behavior is established. Here, sub-swarm behavior is defined as the positions $q_{i,p}^t$ within a quantum orienting their movement $v_{i,p}^t$ towards their shared personal best q_i^* and, depending on the setup, the $n \in [1, SwarmSize - 1]$ global bests. A formula for the update rule can be found in Table 12.

$$q_{i,p}^{t+1} = q_{i,p}^{t} + v_{i,p}^{t+1}$$

$$v_{i,p}^{t+1} = (1-\alpha)v_{i,p}^{t} + \Delta v_{i,p}^{t}$$

$$\Delta v_{i,p}^{t} = \frac{\alpha[q_i^* - q_{i,p}^t] + \sum_{j=1}^{n} \alpha[q_j^* - q_{i,p}^t]}{n+1}$$

Where:

α	= Force of attraction
$(1-\alpha)$	= The inertia for a quantum
n	= Number of informants for a quantum
	(Single QSO: n=1, Fully-Connected QSO: n = swarm size - 1)
q_i^*	= Quantum's best position found
$q_{i,p}^t$	= Quantum's specific position p at iteration t
$v_{i,p}^t$	= Quantum's specific velocity for position p at iteration t
Δv_i^t	= Velocity update rule

Table 12: QSO Update Rules

The loop of the PSO algorithm is mainly left intact in QSO. In this implementation, the only difference occurs at the start of each iteration, before the n global best solution vectors are determined.

At the beginning of each iteration, one *Crossover* and one *Mutation* operator are chosen at random in order to establish new positions in the quanta of the swarm. First, for every GA operation, one quantum is selected to be modified based on its current performance. A specific position in the quantum is chosen as a parent as described in paragraph "Quantum Position Selection". Depending on the operator in use, positions from other quanta might get chosen as a modification vector. The operators used are described in "Crossover" and "Mutation". Afterwards, the offspring generated by the GA operators is established as a new position in the quantum. The process is explained in "Quantum Position Splitting".

At the end of each iteration, the lifespan of quantum positions is updated as described in "Quantum Position Decay".

Quantum Position Selection Every position $q_{i,p}^t$ in a quantum has a lifespan $l_{i,p}^t$ associated. The lifespan denotes how likely the quantum is to be at a certain position. Therefore, the sum of all lifespans of a quantum equals 1. For selection, a roulette-wheel approach based on the respective position's lifespan is used.

Crossover For Crossover, the operators *BinomialCrossover*, *XPointCrossover* and *VotingCrossover* are used. The use of *ChromosomeCrossover* is omitted for its limited exploratory value.

The crossover operator produces two offsprings exactly as described in Chapter 4.2.

Mutation All the mutation operators described in Chapter 4.2 are used. Contrary to crossover, there were some modifications needed, in order for the operators to work.

First of all, since simulation length in pso based algorithms might not reflect the actual length of the candidate solution (see Chapter: 4.3), the actual measured simulation length is used as an upper bound for new values.

Secondly, similar to crossover, a mutation operator yields two resultants. This is realized by duplicating the position's solution vector. Afterwards, both vectors will be iterated simultaneously. Whenever a value in the first vector is modified, the value in the second vector will remain the same and vice versa. Therefore, since changes to specific parameters might end up altering the tellability of the candidate solution drastically, for every parameter that has changed in the first vector, we end up with a second vector, where the parameter still has the same value.

Additionally, since more parameters will be modified, exploration of the mutation operation is significantly increased.

Quantum Position Splitting After the appliance of a GA operator to a quantum position P_1, two new positions are generated. The quantum will assume the better performing of the two as a new position P_2 in the search space. P_2 will copy the velocity vector of P_1 and the lifespan of P_1 will be split equally between P_1 and P_2.

Quantum Position Decay After the quantum swarm's movement phase, the lifespans for all positions are updated. Starting with the worst performing position, each will lose half their current lifespan and immediately adding the amount to the next best performing one. Thus, if a quantum position performs strictly better than others, the specific position will absorb some amount of the lifespan of all lower performing positions every iteration.

If the lifespan for a specific position deceeds a certain threshold, it will vanish adding all its associated lifespan to the next better performing position. In this implementation, I chose the threshold to be $\frac{1}{2^S}$, with S = $\frac{SwarmSize}{2} - 1$. As a result, the amount of positions in the swarm should remain steady, as long as there are particles performing strictly better than others.

Therefore, two new quantum positions enter the swarm in every iteration by appliance of GA operators. Contrary to the classic particles of PSO, quantum-particles can appear at another position in the search space without relying on velocity in order to get there.

While the mechanics behind *Crossover* and *Mutation* remain roughly the same, the intention is quite different. In the GA, these operators are the primary source for population optimization. In QSO, the genetic operations are used for exploration with subsequent optimization via PSO movement rules. Since *Crossover* and *Mutation* are not used for convergence to a particular solution, a higher probability for the operators can be assigned. In this implementation, I used

$$MutationProbability = CrossoverProbability = \frac{\sqrt{|D|}}{|D|}$$

where D is the dimensionality of the problem. As a result, the amount of modifications applied to quantum positions scales with the size of the solution vector in a logarithmic manner.

This exploration strategy is very expensive in terms of computation time. In an attempt to avoid running multiple simulations for bad performing individuals, we only keep the better performing offspring of GA operations.

Since the algorithm keeps generating new individuals, I decided to slightly reduce the initial population size from 20 to 16. There will be 5 different configurations for the statistical analysis: The two better performing setups of the static parameter PSO with $\alpha = 0.1$ and the 3 setups for the gravitational PSO.

4.4.3. Results

QSO	Best	Mean Best	Variance	Mean Average
Single 0.1	0.3	0.248	0.0012	0.175
Full 0.1	0.286	0.242	0.0008	0.164
Single Float	0.3	0.262	0.0008	0.186
Gravity Search	0.3	0.243	0.0015	0.164
Full Float	0.265	0.221	0.0006	0.165

Table 13: Results of the QSO Algorithm

QSO performed strictly better than PSO in every category. The setup of single informant strategy with the gravitational force function yielded the best overall results with the highest average throughout the statistical analysis. In Table 14 are all the p-values, as provided by a t-test, for the comparison between PSO and QSO for each setup respectively.

Setup	P-Value
Single 0.1	0.09
Full 0.1	0.39
Single Float	0
Gravity Search	0
Full Float	0.01

Table 14: PSO - QSO T-Test

While the differences between PSO and QSO using static parameters are not statistically significant, the results when using the gravitational velocity update formula are. Therefore I conclude, the gravitational formula in use is actually feasible and combining gravitational PSO with GA operators seems to solve at least one of the hypothetical problems I mentioned in 4.3.5.

5. Discussion

5.1. Evaluation of Results

An overview of the results for the algorithms analyzed as part of this thesis can be found in Table 18 in the appendix.

In order to compare the performance of the algorithms, an analysis of variance is applied, comparing the results of the algorithms with their respective best performing setup. The results are displayed in Table 15.

Summary of Data				
	Best GA	Best PSO	Best QSO	Total
N	10	10	10	30
$Sigma^X$	2.58	2.377	2.623	7.58
Mean	0.258	0.2377	0.2623	0.253
$Sigma^{X^2}$	0.6786	0.5736	0.6959	1.9481
Std.Dev.	0.0379	0.0309	0.0297	0.0337
Result Details				
Source	SS	df	MS	
Between Groups	0.0035	2	0.0017	F = 1.58208
Withing Groups	0.0295	27	0.011	
Total	0.0329	29		

Table 15: ANOVA Results

Although there are slight differences in the mean best for the algorithms, a p-value of 0.224 indicates that there are no statistical differences in between groups.

Due to limitations in computational capacity, trials per configuration of the algorithms were limited to ten. Consequently, the statistical expressiveness of the results is rather small. Even if the differences in between groups would be significant, they would only yield an indication which could be interpreted for further testing.

In order to derive more reliable results, a more extensive statistical analysis should be done, ideally using multiple different storyworlds for comparison.

After reviewing the results of the algorithms, the simulations for the best results found were reperformed. However, as mentioned in Chapter 3.2.3, the tellability of a certain candidate solution might vary between simulations.

The highest-rated generated story, that reliably results in the same tellability value, has a score of 0.301. The corresponding plot graph can be found in Figure 4. The specific computational values for the aesthetic principles can be found in Table 16 below.

Principle	Absolute Score	Balanced Score
Polyvalence	0.0	0.0
Symmetry	0.225	0.45
Opposition	0.211	0.421
Suspense	0.333	-

Table 16: Tellability Score of Best Plot

In the generated story, Robinson will go to the island. Unfortunately, when he gets hungry, the torrential rain detains him from finding food. When he finally finds some, the food gets stolen. As a result, the agent dies of hunger.

While this might not be a story, that sounds interesting to a human listener, the plot does manage to get rated very well by the current implementation of the tellability function. Since the character has only very few goals in mind throughout the simulation, his intentions are very *symmetrical*. The *Opposition* and *Suspense* values are high due to the agent failing to achieve his goals.

In addition to the scheduling of happenings, the appropriate personality also must be found by the algorithms. Even small changes to a personality parameter can result in detrimental changes to the overall tellability score.

During the review of the results I noticed, roughly half of the best solutions found show inconsistencies. When running a few hundred simulations, there will be some simulations, where the decision making of the agent will be delayed in a certain step (see Chapter 3.2.3). Sometimes these inconsistencies might lead to a very good result compared to the rest of the population. These inconsistencies do no necessarily depict plots, that are impossible to generate under normal circumstances, but ones that can be achieved by finding the correct parameter combination. Unfortunately, when one of those high scoring plot is found due to an inconsistency, new individuals will be based on the corresponding parameters. The actual average mean bests for the best setups of the algorithms are shown in Table 17.

Algorithm	Mean Best
GA	0.232
PSO	0.146
QSO	0.149

Table 17: Actual Overall Results

The mean bests, particularly for the PSO- based algorithms, are significantly lower, than the reported scoring. When in PSO a candidate solution gets a very high rating, all particles might move towards the respective position. The algorithms will continue to search at the corresponding position in the search space, even if there is nothing to be found or only a slight chance for a particular simulation to actually return a good result.

Therefore it is even more notable, that some of the algorithms managed to find good results, while only being able to look at a limited amount of the neighborhood in a discontinuous infinite search space.

5.2. Additional Use Cases

At the start of this chapter I want to ensure, that the content is not meant to discredit any work that has been put into developing the Robinson storyworld by any means. Different algorithms may have different prerequisites in order to work in a specific domain.

Nature-inspired algorithms can be very demanding towards the systems they are applied to. In the context of the InBloom storytelling framework, they have proven to reliably find combinations for parameters, that result in an increase in the tellability score. Therefore, additional applications for nature-inspired algorithms arise naturally. The results may vary depending on the objective fitness function used.

This Chapter is divided into two sections. At the beginning of the study phase of this thesis, the tellability function was not balanced. Therefore, repetitive plot structures were rated higher and different plot structures were found to be the best story in the environment. Chapter 5.2.1 will elaborate on how the algorithms were used in order to find bugs such as logical loopholes in the environment.

Subsequently, Chapter 5.2.2 will describe the usage of nature-inspired algorithms, after the debugging process was finished.

5.2.1. Debugging

As previously mentioned, at the beginning of the work with the Robinson storyworld, the fitness function was not balanced. Whenever there was an option for the algorithms to come up with a solution, that resulted in infinite repetitions of the same action, the score for *Suspense* and *Symmetry* would be very high. As a result, these cyclical plot structures were rated very well by the fitness function. However, the concept of agents dying was not yet implemented and, hence, the agent would not die while being stuck in an infinite cycle.

Robinson Building Infinite Huts When Robinson gets tired, the plan of to sleep is initiated. According to the description in Chapter 3.1.3, he will first build a hut and only go to sleep thereafter.

In the initial version of the environment, the action *buildHut()* updated the believes of any agent on the island, that there now is a hut. The algorithms quickly "realized", when Robinson never gets to the island, because he would not have the appropriate personality to go on the cruise in the first place, his believes would not be updated. He then could build huts indefinitely.

In order to fix this, actions, meant to be executed only on the island, were restricted to take place on the island only.

Hungry Robinson At some point, all possibilities to create infinite cycles were fixed in the environment. The algorithms then "realized", that there were two different frameworks interacting with each other. There was a bug in the interaction of the systems found, that could be used in order to mess with Robinson's reasoning cycle.

When Robinson gets hungry, he first needs to have food in his inventory in order to eat it. He will therefore first execute the action *findFood()*, which updates his set of believes to represent him having food. Subsequently, the plan *eatFood* is initiated, resulting in the agent eating the food in his inventory. The algorithms found, if the *FoodStolenHappening* occurs exactly at the point in time, where the believe is updated but the action *eatFood* is not executed yet, the agent will not realize, that his food was stolen. In every subsequent simulation step, the agent will try to eat, but fail, because there is no food in his inventory.

The issue was fixed by inserting a catch-phrase in every action, that has a believe associated. By adding this catch-phrase, the believe system will be updated and plans will be adjusted accordingly.

Robinson eating non-existent food Contrary to the previous examples, this bug has nothing to do with infinite cycles. While looking at the plot graphs it can be observed, that Robinson was able to eat, even though his food was stolen previously. It turned out, that the function for retrieving items from the inventory was a not working as intended on the branch in GitHub.

5.2.2. Tellability Space Optimization

After finishing the debugging process, the fitness function on the GitHub branch was updated correspondingly. As a result, no cyclical plot structures were deemed to be optimal anymore. Having a new version of the function determining the plot always results in a new cycle of revising the results returned by the algorithms.

This time, the conceptual space itself was optimized in order to favor more interesting stories. A few examples are given below.

Robinson the firefighter After updating the tellability function, *Opposition* was introduced. The *FireHappening* was a good method, to increase the *Opposition* score in a story. The best story that could be found was sending Robinson to the island, letting him perform *extinguishFire()* and rescue him subsequently.

There and Back Again, Robinson's Holliday After introducing the plan *!rescueSelf* and the action *swimToIsland* for better readability of the plot graph, the best story that could be found was for robinson to get stranded and then immeadiatly getting rescued. Therefore *Symmetry*, *Suspense* and *Opposition* were rated very highly in the plot graph.

In order to fix both previous examples, the ship rescue happening was set to only occur after the 5. step. Therefore, a minimal plot length was introduced and, hence, plots with a length of 2-3 steps were not preferred anymore.

In fact, the current best plot of Robinson going to the Island and basically starving to death could be treated the same way. If one would deem this story not to be desirable and to generate a plot, that is more appreciable by humans, there could be similar measures in order to reduce the tellability of this particular plotline.

Studies performed as part of this thesis stopped at this point. The intention was to have a plot with specific requirements - in this case personality structure combined with"disruptive happenings" (see: Chapter 3.1.3) - in order to see if the algorithms could find the specific structure reliably.

5.3. Future Work

First, a more extensive statistical analysis is needed to assess the performance of the algorithms used in this thesis. An in-depth analysis with much more trials per configuration would be able to determine significant differences much more reliably. Ideally, the algorithms should be tested in different storytelling environments aswell.

Supplementary to the more in-depth analysis, more configurations of the algorithms could be tested. Especially for the modified version for the GA and PSO, there are many options to fine-tune the behavior of the algorithm.

Finally, an extension to the general approach could be implemented, where happenings could be scheduled multiple times. Currently, a happening can only be instantiated once per patient. In this implementation, the only way to schedule more than one, would be to hard code a happening as static, or ascribe multiple positions in the solution vector to a particular happening. An idea for an implementation would be for the solution vectors to contain lists instead of numbers. If lists are empty, a happening would be deactivated, otherwise the happenings would be scheduled at the respective time steps, contained in the lists.

References

Aldous, D. and U.U. Vazirani (1994). “”GO with the Winners” Algorithms”. In: *FOCS*, pp. 492–501.

Anton, Roman (2018). “Reevaluation of artificial intelligence engine alpha zero, a self-learning algorithm, reveals lack of proof of best engine, and an advancement of artificial intelligence via multiple roots”. In: *Journal of Mathematical and Theoretical Physics* 1, pp. 22–33.

Bansal, Jagdish Chand and Kusum Deep (2012). “A Modified Binary Particle Swarm Optimization for Knapsack Problems”. In: *Applied Mathematics and Computation* 218.22, pp. 11042 –11061.

Berov, L. (2017b). “Towards a computational measure of plot tellability”. In: *Proceedings of the 10th International Workshop on Intelligent Narrative Technologies*, 169–175.

Berov, L. and K.U. Kühnberger (2018). “An evaluation of perceived personality in fictional characters generated by affective simulation”. In: *Proceedings of the Ninth International Conference on Computational Creativity, Salamanca, Spain.*

Boden, M (1990). “The creative mind: Myths and mechanisms”. In.

Bordini, Rafael H. and Jomi F. Hübner (2007). *Jason - A Java-based interpreter for an extended version of AgentSpeak.*

Bowden, B. V. (1953). *Faster Than Thought: A Symposium on Digital Computing Machines.* USA: Pitman Publishing, Inc.

Chrominski, Kornel, Magdalena Tkacz, and Mariusz Boryczka (2020). “Epigenetic Modification of Genetic Algorithm”. In: *Computational Science – ICCS 2020.* Springer International Publishing, pp. 267–278.

Dartnall, T. (Ed) (1994). *Artificial intelligence and creativity: An interdisciplinary approach.* Springer.

DeFoe, Daniel (1719). London: W.Taylor.

Djannaty, Farhad and Saber Doostdar (2008). “A Hybrid Genetic Algorithm for the Multidimensional Knapsack Problem”. In: *Int. J. Conemp. Math. Sciences* 3.9, pp. 443 –456.

Dorigo, M., M. Birattari, and T. Stutzle (2006). “Ant colony optimization”. In: *IEEE Computational Intelligence Magazine* 1.4, pp. 28–39.

Du, Yu et al. (2010). “Improved Quantum Particle Swarm Optimization by Bloch Sphere”. In: vol. 6145, pp. 135–143.

“Exploring the fascinating world of language” (1963). In: *Business Machines XLVI(3).*

Fister Jr., Iztok et al. (2013). *A Brief Review of Nature-Inspired Algorithms for Optimization.* arXiv: `1307.4186 [cs.NE]`.

Foster, E. M. (1927). “Aspects of the novel: The timeless classic on novel writing”. In: *Library of Congress Cataloging-in-Publication Data.*

Gervás, Pablo (2009). “Computational Approaches to Storytelling and Creativity”. In: *AI Magazine* 30, pp. 49–62.

Green, Christopher (2005). “Was Babbage’s Analytical Engine Intended to Be a Mechanical Model of the Mind?” In: *History of Psychology - HIST PSYCHOL* 8, pp. 35–45.

Hashimoto, D.A. et al. (2019). “Artificial Intelligence in Surgery: Promises and Perils”. In.

Kashyap, Ramgopal (2019). “Artificial Intelligence Systems in Aviation”. In: pp. 1–26.

Kennedy, James and Eberhart Russel (1995). “Particle Swarm Optimization”. In: *Proceedings of ICNN’95 - International Conference on Neural Networks.* Vol. 4, 1942–1948 vol.4.

Labov, W. (1972). “Language in the inner city: Studies in the black english vernacular”. In: *University of Pennsylvania Press.*

Langohr, Laura (2011). “Virtual Story Generation: from TALE-SPIN to the Virtual Storyteller”. In: *Seminar on Computational Creativity.*

Lehnert, W. (1981). “Plot Units and Narrative Summarization”. In: *Cogn. Sci.* 5, pp. 293–331.

McCormack, J. and M. (Eds.) d’Inverno (2012). *Computers and creativity.* Springer.

McCrae, R. R. and O. P. John (1992). “An introduction to the five-factor model and its applications”. In: *J. Pers 60(2)*, 175–215.

Meehan, James R. (1977). “TALE-SPIN, an Interactive Program That Writes Stories”. In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 1.* IJCAI’77. Cambridge, USA: Morgan Kaufmann Publishers Inc., 91–98.

Mehrabian, A. (1996). “Analysis of the big-five personality factors in terms of the pad termperament model”. In: *Aust J Psychol 48 (2)*, pp. 86–92.

Mohd Sabri, Norlina, Mazidah Puteh, and Mohamad Rusop (2013). “A review of gravitational search algorithm”. In: *International Journal of Advances in Soft Computing and its Applications* 5.

Pant, Millie, Radha Thangaraj, and Ajith Abraham (2008). “A new quantum behaved particle swarm optimization”. In: pp. 87–94.

Peri, Daniele, Matteo Diez, and Giovanni Fasano (2012). “Comparison between Deterministic and Stochastic formulations of Particle Swarm Optimization, for Multidisciplinary Design Optimization”. In.

Poli, Riccardo, James Kennedy, and Tim Blackwell (2007). “Particle Swarm Optimization: An Overview”. In: *Swarm Intelligence* 1.

Rao, Anand and Michael Georgeff (Jan. 2001). “Modeling Rational Agents within a BDI-Architecture”. In: *Proceedings of the international conference on principles of knowlegde representation and reasoning.*

Rao, Anand S. (1996). “AgentSpeak(L): BDI agents speak out in a logical computable language”. In: *Agents Breaking Away.* Springer Berlin Heidelberg, pp. 42–55.

Rashedi, E., H. Nezamabadi-pour, and S. Saryazdi (2009). “GSA: A Gravitational Search Algorithm”. In: *Information Sciences* 179.13, pp. 2232–2248.

Reeves, Colin (1997). “Genetic Algorithms for the Operations Researcher.” In: *INFORMS Journal on Computing* 9, pp. 231–250.

“Robot With Mechanical Brain Thinks up Story Plots” (1931). In: *Popular Mechanics.*

Ryan, James (2017). “Grimes’ Fairy Tales: A 1960s Story Generator”. In: *Interactive Storytelling.* Springer International Publishing, 89–103.

Ryan, M.-L. (1991). “Possible Worlds, Artificial Intelligence, and Narrative Theory”. In: *Indiana Pennsylvania Press.*

Stützle, Thomas and Marco Dorigo (1999). *ACO Algorithms for the Traveling Salesman Problem.*

Sun, Jun, Bin Feng, and Wenbo Xu (2004). “Particle swarm optimization with particles having quantum behavior”. In: vol. 1, 325 –331 Vol.1.

Tian, Dongping and Zhongzhi Shi (2018). “MPSO: Modified particle swarm optimization and its applications”. In: *Swarm and Evolutionary Computation* 41, pp. 49 –68. ISSN: 2210-6502.

Tsujimoto, Takahiro, Takuya Shindo, and Kenya Jin’no (2011). “The neighborhood of canonical deterministic PSO”. In: pp. 1811–1817.

Valls-Vargas, Josep, Jichen Zhu, and Santiago Ontanon (2017). “From computational narrative analysis to generation: a preliminary review”. In: pp. 1–4.

Wiggins, G.A. (2006). “Searching for computational creativity”. In: *New Generation Computing, Computational Paradigms and Computational Intelligence, 24*, pp. 209–222.

Wilke, Sven and Leonid Berov (2018). “Functional Unit Analysis: Framing and Aesthetics for Computational Storytelling”. In.

Wippermann, Julia (2020). *Using Reinforcement Learning to Improve the Tellability of Computationally Generated Plots.*

Yang, Xin-She (2020). “Nature-inspired optimization algorithms: Challenges and open problems”. In: *Journal of Computational Science* 46, pp. 101–104.

Yang, Xin-She and Mehmet Karamanoglu (2013). “Swarm Intelligence and Bio-Inspired Computation: An Overview”. In: *Swarm Intelligence and Bio-Inspired Computation*, pp. 3–23.

A. Appendix

Setup	Best	Mean Best	Variance	Mean Average
GA Base Version				
0.25 / 0.15	0.236	0.218	0.0001	-
0.2 / 0.1	0.312	0.258	0.0013	-
GA Floating Parameter Version				
0.1	0.319	0.2434	0.0015	-
0.05	0.309	0.2375	0.0016	-
PSO Base Version				
Single 0.1	0.281	0.227	0.0008	0.144
Single 0.05	0.281	0.218	0.0017	0.128
Full 0.1	0.3	0.238	0.0009	0.159
Full 0.05	0.272	0.214	0.0011	0.135
PSO Gravitational Version				
Single	0.269	0.211	0.0014	0.157
Gravity Search	0.24	0.195	0.001	0.139
Full	0.247	0.191	0.0009	0.138
QSO				
Single 0.1	0.3	0.248	0.0012	0.175
Full 0.1	0.286	0.242	0.0008	0.164
Single Float	0.3	0.262	0.0008	0.186
Gravity Search	0.3	0.243	0.0015	0.164
Full Float	0.265	0.221	0.0006	0.165

Table 18: Results of all Nature-Inspired Algorithms

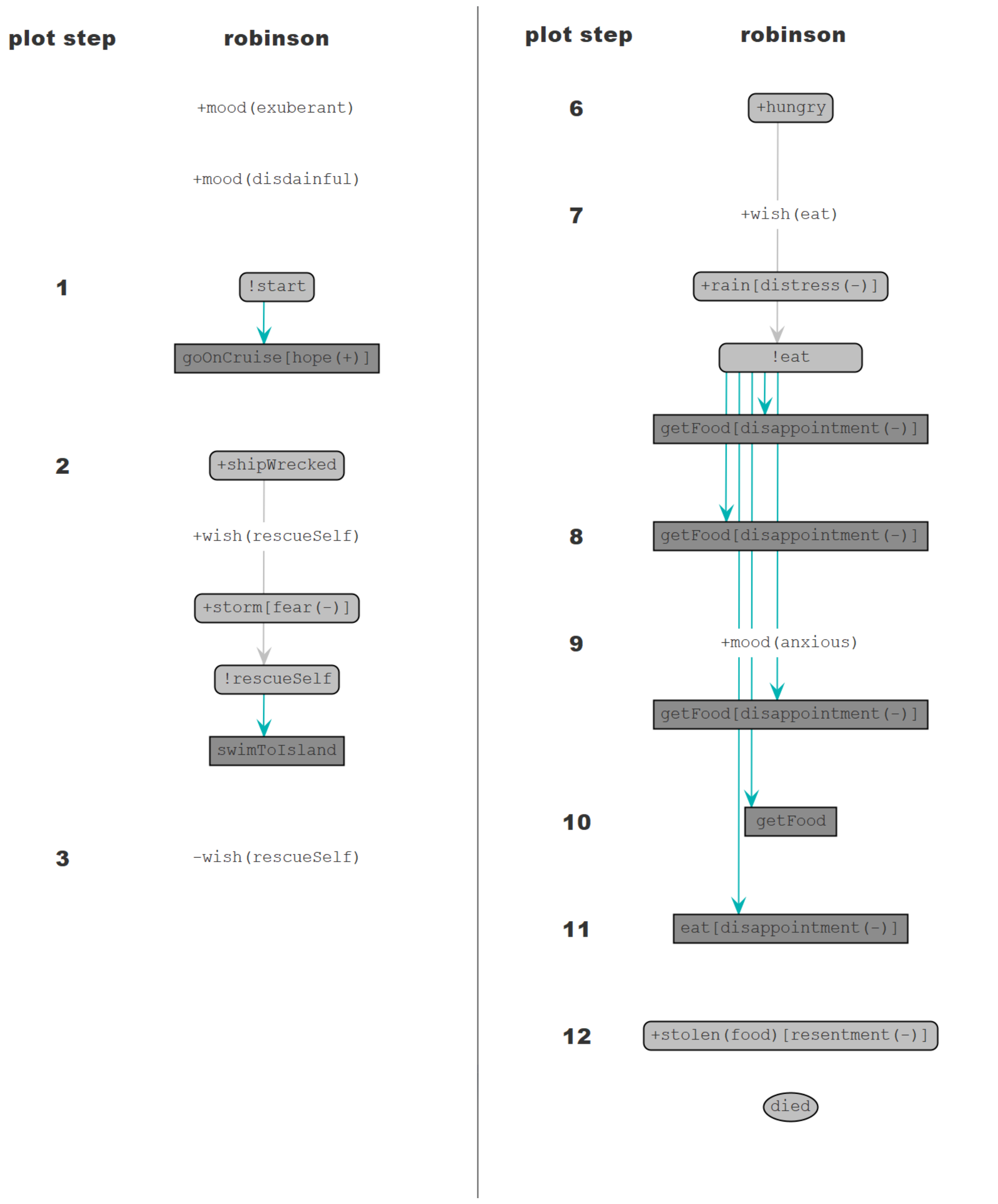


Figure 4: Best Plot Found