SenseAct


Luís Filipe Pookatham Ribeiro Gomes

Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering

Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues
Member of the Committee: Prof. João Coelho Garcia

November 2019
Acknowledgments

This work would not be possible without the caring and dedication of my family. Starting with my father Vasco Gomes and my grandmother Maria Luísa Gomes, who have supported me since the beginning of my education, I would like to give them the utmost gratitude I can possibly provide. I would also like to thank the rest of my family for their friendship and highlighting the best of me.

For my girlfriend Clara Vieira, and her parents Claudia Fonseca and António Gouveia, I want to give them a special acknowledgement. I would like to thank them for the constant good moments we have being experienced and for being present in the other moments. I cherish everything we have done together. For that reason, I consider them as part of my family.

To my friends, I would like to thank them for accepting me as I am, for being able to open up with me, for every invitation they gave and for every joyful moments we have spent.
Abstract

This thesis tackles the problem of agents that need to decide which actions to take based on incomplete information, i.e., assuming that they cannot be fully aware of their context. In particular, their awareness of the surrounding environment is based on measurements that can be requested on demand, and where there is a cost associated both with the measurement process and with the quality of the decision. For that purpose, a model called Measurement-based Decisions has been made to set what is common in use cases with that same problem. Moreover, a learning agent has been conceptualized to follow the model, and solve some of the challenges it presents. This agent, called SenseAct, has a simulative learning phase where it creates a cost table through dynamic programming and genetic algorithms, and a real-time decision phase, where it chooses one of several groups of actions from the cost table through a probability distribution table. As a proof of concept, it presents better results, in performance and in quality, than straw men such as the greedy and the lazy approaches.

Keywords

Emergency Management; Stochastic Optimization; Uncertainty; Reinforcement Learning; Genetic Algorithm; Dynamic Programming.
Resumo

Esta tese aborda um problema onde há agentes que precisam de decidir com base em informações incompletas, ou seja, assumindo que eles não podem estar totalmente cientes de seu contexto. Em particular, a sua conscientização sobre o ambiente circundante é baseada em medições que podem ser solicitadas no momento e onde há um custo associado ao processo de medição e à qualidade da decisão. Para esse fim, foi criado um modelo chamado Measurement-based Decisions ("Decisões Baseadas em Medições") para definir o que é comum nos casos de uso. Para além disso, um agente de aprendizagem foi conceptualizado para seguir o modelo e resolver alguns dos desafios que ele apresenta. Esse agente, chamado SenseAct, possui uma fase de aprendizagem, na qual cria uma tabela de custos a partir de programação dinâmica e de algoritmos genéticos, e uma fase de decisão em tempo real, onde escolhe um de vários grupos de ações da tabela de custos a partir de uma tabela de distribuição de probabilidade. Como prova de conceito, apresenta melhores resultados, em termos de desempenho e de qualidade, do que abordagens como as abordagens greedy e lazy.

Palavras Chave

Gestão de Emergências; Optimização Estocástica; Incerteza; Aprendizagem Reforcada; Algoritmo Genético; Programação dinâmica.
# Contents

1 Introduction .................................................. 1
   1.1 Motivation ............................................... 3
   1.2 Contributions .......................................... 4
      1.2.1 Measurement-based Decisions ..................... 4
      1.2.2 SenseAct .......................................... 4
   1.3 Organization of the Document ......................... 5

2 Background ................................................... 7
   2.1 Motivation Examples ..................................... 9
      2.1.1 Adversarial Submarine ............................ 9
      2.1.2 Building Fire .................................... 10
      2.1.3 Cancer Patient .................................. 10
   2.2 Measurement-based Decisions ......................... 10
      2.2.1 Waiting Time .................................... 12
      2.2.2 Waiting Cost & Deployment Cost ................. 12
      2.2.3 Detection Range ................................ 13
      2.2.4 Decaying Speed ................................ 13
      2.2.5 Prediction Distribution ......................... 14
   2.3 Summary ................................................ 15

3 Architecture .................................................. 17
   3.1 Possible Approaches .................................... 19
      3.1.1 Optimal Agent .................................... 19
      3.1.2 Greedy Reflex Agent ............................. 19
      3.1.3 Lazy Reflex Agent ............................... 20
      3.1.4 Monte Carlo Tree Search (MCTS) Learning Agent . 21
   3.2 SenseAct ................................................. 22
      3.2.1 Performance Element ............................. 23
         3.2.1.A Sensor Table ................................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1.B</td>
<td>Area Table</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1.C</td>
<td>Measurement Table</td>
<td>25</td>
</tr>
<tr>
<td>3.2.1.D</td>
<td>Cost Table</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Critic</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2.A</td>
<td>Intervals and Weights</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2.B</td>
<td>Mean Waiting Cost (MWC)</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2.C</td>
<td>Mean Subsequent Collateral Cost (MSCC)</td>
<td>29</td>
</tr>
<tr>
<td>3.2.2.D</td>
<td>Batch Deployment Cost (BDC)</td>
<td>29</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Problem Generator</td>
<td>30</td>
</tr>
<tr>
<td>3.2.3.A</td>
<td>First Generation</td>
<td>30</td>
</tr>
<tr>
<td>3.2.3.B</td>
<td>Next Generations</td>
<td>31</td>
</tr>
<tr>
<td>3.2.3.C</td>
<td>Evaluation using Critic</td>
<td>33</td>
</tr>
<tr>
<td>3.2.3.D</td>
<td>Stopping Condition</td>
<td>33</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Ahead-of-time (AOT) Learning Element</td>
<td>33</td>
</tr>
<tr>
<td>3.2.4.A</td>
<td>Well-Known Collateral Costs</td>
<td>34</td>
</tr>
<tr>
<td>3.2.4.B</td>
<td>Useful Interval Groups</td>
<td>35</td>
</tr>
<tr>
<td>3.2.4.C</td>
<td>Endpoint Groups</td>
<td>36</td>
</tr>
<tr>
<td>3.2.4.D</td>
<td>GA-based Batches</td>
<td>36</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Just-in-time (JIT) Learning Element</td>
<td>37</td>
</tr>
<tr>
<td>3.2.5.A</td>
<td>Probability Mass Function (PMF)</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Summary</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Programming Language: Python</td>
<td>41</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Parallelism Limitation: Global Interpreter Lock (GIL)</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Dependencies</td>
<td>41</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Alternative Dependency: Distributed Evolutionary Algorithms in Python (DEAP)</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Other Settings</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Evaluation Environment</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>AOT Learning Element Performance</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance Element Real-Time Constraints</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>SenseAct versus Straw Men</td>
<td>48</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Waiting Cost</td>
<td>48</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Deployment Cost</td>
<td>48</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Overall Cost</td>
<td>49</td>
</tr>
</tbody>
</table>
5.5 Summary ................................................................. 49

6 Discussion ...................................................... 53
6.1 Result Interpretations ................................................. 55
6.2 Future Work .......................................................... 55
   6.2.1 Algorithmic Optimizations ..................................... 55
      6.2.1.A Other Genetic Algorithms (GAs) ................................. 55
      6.2.1.C Predictive analytics in learning elements ....................... 55
   6.2.2 Further generalization of Measurement-based Decisions ............... 56
      6.2.2.A Deployment Time .................................................. 56
      6.2.2.B Deployment Fault & Placement Fault ............................. 56
      6.2.2.C Placement Fault ................................................. 57
      6.2.2.D Decaying Fault .................................................. 57
      6.2.2.E Prediction Fault ................................................. 57
      6.2.2.F Detection Range with PMF ...................................... 57
      6.2.2.G Dynamic sensitive areas ....................................... 57
      6.2.2.H Byzantine Fault Tolerance in Measurement-based Decisions ....... 58
   6.3 Summary .......................................................... 58

7 Conclusion .......................................................... 61
List of Figures

2.1 Overview of Measurement Process ........................................... 11

3.1 Optimal Agent - Component Diagram ........................................ 19
3.2 Greedy Reflex Agent - Component Diagram ................................. 20
3.3 Lazy Reflex Agent - Component Diagram .................................... 20
3.4 MCTS Learning Agent - Component Diagram ................................ 22
3.5 SenseAct Learning Agent - Component Diagram ......................... 23
3.6 SenseAct Performance Element - Activity Diagram ..................... 24
3.7 SenseAct Critic - Activity Diagram ........................................... 27
3.8 SenseAct Problem Generator - Activity Diagram ......................... 30
3.9 SenseAct AOT Learning Element - Activity Diagram .................... 34

5.1 AOT Learning Element Performance - Box and Whisker Chart .......... 46
5.2 Performance Element Performance - Box and Whisker Chart ............ 47
5.3 Performance Element Waiting Cost Quality - Box and Whisker Chart .... 49
5.4 Performance Element Deployment Cost Quality - Box and Whisker Chart 50
5.5 Performance Element Overall Cost Quality - Box and Whisker Chart .... 51
List of Tables

2.1 PMF of incident $x$ when time $t = 0$ ........................................ 14
2.2 PMF of incident $x$ when time $t = 1$ ........................................ 14
2.3 PMF of incident $x$ when time $t = 2$ ........................................ 14

3.1 Adversarial Submarine Sensor Table ........................................ 25
3.2 Adversarial Submarine Area Table ........................................... 25
3.3 Measurement Table $t = 1\text{min}$ ........................................... 25
3.4 Arriving Measurements when $t = 1\text{min}$ .................................. 26
3.5 Example of a Cost Table .......................................................... 26
3.6 PMF when $t = 1\text{min}$ .......................................................... 38

List of Algorithms

3.1 First Generation Strategy ....................................................... 31
3.2 Next Generations Strategy ...................................................... 32
3.3 Stopping Condition Strategy .................................................... 33
3.4 Well-Known Collateral Costs Strategy ....................................... 35
3.5 Useful Interval Groups Strategy ............................................... 36
3.6 Aggregation by Similarity Strategy .......................................... 37
3.7 GA-based Batches Strategy ..................................................... 37
Listings
Acronyms

AIMA  Artificial Intelligence: A Modern Approach
AOT   Ahead-of-time
BDC   Batch Deployment Cost
DEAP  Distributed Evolutionary Algorithms in Python
GA    Genetic Algorithm
GIL   Global Interpreter Lock
GPS   Global Positioning System
IDE   Integrated Development Environment
JIT   Just-in-time
LTS   Long Term Support
MCTS  Monte Carlo Tree Search
MSCC  Mean Subsequent Collateral Cost
MWC   Mean Waiting Cost
PMF   Probability Mass Function
RAM   Random-Access Memory
SURTRAC Scalable Urban Traffic Control
Introduction

Contents

1.1 Motivation ................................................................. 3
1.2 Contributions ............................................................. 4
1.3 Organization of the Document ........................................ 5
This thesis proposes SenseAct, an optimization framework that continuously deploys sensors in uncertain environments. It uses previously taken measurements and planning instances that define reactive decisions through reinforced learning and stochastic optimization. Throughout this introductory chapter, a range of Motivation is presented to justify its conception, followed by the Contributions made while developing it. In the end of the chapter, the implementation of SenseAct is summarized in the Organization of the Document section.

1.1 Motivation

This thesis tackles the problem of agents that need to decide which actions to take based on incomplete information, i.e., assuming that they cannot be fully aware of their context. In particular, their awareness of the surrounding environment is based on measurements that can be requested on-demand, and where there is a cost associated both with the measurement process and with the quality of the decision.

Making decisions that cannot be context-aware without taking measurements over a given environment presents a multitude of factors that bring uncertainty over the reliability of the decisions. For instance, the size of the physical or virtual space where the measurements are to be taken raises the question of where to deploy the sensors that will take the measurements, while the mutability of the environment over time may decay previous measurements and force their revalidation through new measurements. Also, the unreliability of physical sensors, which are sometimes made of complex electromagnetic components, makes them prone to failure and external manipulation, so their redundancy should also be taken into account. All these considerations should be made so that the cost of not knowing the context is minimized without making useless measurements since they also involve some cost.

A valid solution to this problem involves permanent surveillance, where a range of sensors periodically but constantly broadcasts information to have a strategic overview of the environment. Scalable Urban Traffic Control (SURTRAC) [1], for instance, extends already installed signal control with multi-agent planning to reduce travel times and vehicle emissions. However, in the kind of deployments we are targetting, the surveillance infrastructure does not exist, and not reasonable to be implemented. Considering the Adversarial Submarine example, to detect an adversarial submarine over a much larger sea would involve a proportionally large amount of sensors that may decay and be posteriorly replaced over time (section 2.2.1). In the Building Fire example, the number of buildings, their complexity, and privacy issues difficult reliable and complete surveillance. And, in the Cancer Patient example, ethic and welfare concerns arise when continuously probing a patient.
1.2 Contributions

The following contributions have been made while elaborating this thesis:

1.2.1 Measurement-based Decisions

The aforementioned challenges have been grouped under a model called *Measurement-based Decisions*. In common, these events share a few intertwined characteristics that require continuous decision making over future input. Abridging section 2.2, they share a set of common characteristics: firstly, they usually have at least an incident, which is a target to locate or a value to identify, and at least a range of locations or values with a level of importance that involves a cost if there is a suspicion that the incident is within it; secondly, given the size, opacity or complexity of the environment, its reduced visibility prevents a precise detection of incidents without resorting to sensor-based measurements over time; thirdly, the uncertainty over movement of an incident forces us to constantly update our measurements, and the uncertainty over the sensors themselves, which can also be faulty, may lead to wrongful measurements, either accidentally or maliciously.

1.2.2 SenseAct

Besides the model, this thesis details a framework that applies its logic called *SenseAct*. Its functionality is split in a planning phase, where a strategy that involves taking measurements over some time is planned backward in time, and an execution phase, where the strategy is implemented to minimize the cost generated by the incident. The strategy is structured into a *cost table* through dynamic programming over Genetic Algorithms (GAs), and is completed with information from a probability distribution table that is updated using external input.

For the implementation and evaluation of SenseAct, a proof of concept was built using python without resorting to any framework in particular.

As a use case, this thematic is inserted across intelligent system fields, although it can be extended to be also considered in distributed system fields. For instance, the problem itself is an optimization one, where the overall cost of the incident should be minimized. In terms of architecture, SenseAct is an active reinforced learning agent with planning features, most specifically contingency planning on the planning phase and online replanning on the execution phase, as well as searching capabilities in the planning phase: local search and adversarial search. In both phases, it applies probabilistic reasoning over time.
1.3 Organization of the Document

This document is divided into the following chapters:

- The **Background** chapter analyses the environment where the framework would be used on, where a motivating example of an adversarial submarine is given to identify characteristics and problems. Some other use cases are mentioned to show how generalized SenseAct can be.

- Then, an architecture that seeks to solve those problems is proposed in the **Architecture** chapter, detailing it in a top-down approach, through diagrams, formulas, and examples. Alternative approaches are presented with an explanation of why they are not used by SenseAct. Except for the Monte Carlo approach, the remaining approaches are used as baselines later on the **Evaluation** chapter.

- In the **Implementation** chapter, an implementation of SenseAct is represented in pseudocode, with notes about its development and technologies that were used.

- In the **Evaluation** chapter, a description of the environment used to evaluate SenseAct and the previously mentioned straw men are made, followed by the settings used to generate the results and the results themselves.

- In the **Discussion** chapter, the results are interpreted and justified by identifying limitations of the prototype. Future work is also mentioned, with possible enhancements in the architecture and implementation, as well as details from Measurement-based Decisions that were not taken into account and that could be studied and tackled in the future.
2 Background

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Motivation Examples</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Measurement-based Decisions</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Summary</td>
<td>15</td>
</tr>
</tbody>
</table>
This chapter presents a model that covers common characteristics and challenges over a range of Motivation Examples. It starts with a motivational example of an adversarial submarine in the middle of a sea, which is referred throughout this thesis and generalizes the model by identifying the aforementioned characteristics in a bottom-up approach. These characteristics are also seen in two other use cases: Building Fire and Cancer Patient.

In addition to what is covered in this chapter, some other aspects of measurement-based decisions were identified but, given the complexity of this topic, not tackled by SenseAct. Instead, they are only mentioned in chapter 6.

2.1 Motivation Examples

2.1.1 Adversarial Submarine

In this imaginary example, an adversarial submarine is hidden in the middle of a sea large enough to not be doable to maintain permanent sensors. Nonetheless, since there are several sensitive areas like military infrastructure which may be of the interest of the submarine to investigate, ephemeral sensors are used to estimate its location. Through their measurements, an incident interval is delimited with positions where the submarine might be. An alert is triggered if the incident interval intersects with any sensitive area.

For simplicity’s sake, the sea is represented as an uni-dimensional line, in between kilometers $[0..100]$. There are two sensitive areas: 1. a less sensitive area in between kilometers $(45..70)$, called yellow area; 2. and a more sensitive area in between kilometers $(40..45)$, called red area. Each sensitive area has an associated alert, which is an action that is triggered to protect the area of any threat that is either visible or suspected to exist within the area. They have a cost over time while they are active; the yellow area has a cheaper yellow alert, which costs 50 dollars per minute, while the red area has a more expensive red alert, costing 1000 dollars per minute. When both the red alert and the yellow alert are being triggered, only the cost of the most expensive alert, in this case, the red alert, is considered.

As future work, the incident interval may consider multiple submarines with arbitrary movements, but, in this thesis, it considers only a single submarine with movements that are limited to a speed of 1 km per minute. The measurements from the various sensors can be combined at any given instant, and are also taken into account as time goes by. On one hand, the incident interval is reduced by intersecting all measurements into a single multi-interval; a single type of sensor called cheap sensor is used with an observational error of 5 kilometers and a cost of 10 dollars per deployment. On the other hand, it grows because the generated measurements lose precision at the same rate as the submarine max speed of 1 kilometer per minute. Moreover, the probability of the submarine being at a given minute and kilometer interval $x \in (0..1), (1..2), \ldots, (99..100)$ is propagated into the probabilities of the adjoining interval at the
following minute. For instance, a third of the probability of the submarine being in (1..2) goes to the probability in (0..1) at the following minute.

2.1.2 Building Fire

On a more dynamic and complex scenario, we would have a fire that is spreading across a building, and incident commanders that would deploy a combination of assessment mechanisms (e.g. drones, fire dogs, firefighters) into the building to determine a panoply of factors (e.g. number of floors on fire, the structural integrity of the building and building materials) that may affect the spreading of the fire. The sensitivity of this incident would also vary over time, from the number of casualties to the number of people requiring rescue, and also how many main rooms like corridors or stairs that are still accessible to the outside.

This information would also be compared and complete external sources like weather reports and city blueprints. Also, permanent surveillance could be installed on the building. However, that is not always the case for economic or privacy reasons.

In all cases, most information would bring uncertainty in many levels from faulty or occluded sensors, and human error to outdated documentation.

2.1.3 Cancer Patient

On a less time-sensitive scenario, a cancer patient could undergo one of many different kinds of treatments, each one with their financial cost. And several testing procedures such as biopsies and tumor marker tests could be made to produce documentation like radiology reports that can help doctors decide which treatment, how long and in which dosage to apply.

One of the challenges in this example is the existence of direct health risks in both treatments and screening techniques. Some of them are available in the form of low-value health care services, which are prevalent in Medicare health plans [2].

Like in the Building Fire case, human error is involved, and some testing procedures are inaccurate or lack specificity or sensitivity. Moreover, some treatments may prove ineffective and can change the overall structure and composition of the body.

2.2 Measurement-based Decisions

In all the Motivation Examples, there is some kind of incident (e.g. submarine, fire or cancer) that moves across an environment (e.g. sea, building, human/animal body). And, due to lack of visibility, there is uncertainty which involves extra cost, either because of lack of decision (e.g. leaving a building in
flames) or lack of evidence to proceed to a better decision (e.g. keep on red alert or chemotherapy). To optimize costs, probing agents (e.g. sea probes, firefighters, tumor marker tests) should be deployed to measure and possibly contain the incident.

In a more mathematical point of view, we have an incident $x$ that is contained and moves within a multidimensional interval $I$. Inside $I$, there are other multidimensional intervals $i_1, i_2, \ldots, i_n$ which represent any kind of sensitive area or entity (e.g. oil platforms, rooms with people or body tissue) have an associated cost $c_1, c_2, \ldots, c_n$. Under extreme uncertainty, the incident may be in any position within $I$, and this uncertainty is represented by an interval $i_x = I$, or incident interval. In this situation, it is assumed that the incident is within the most expensive sensitive area, so its cost is assumed as well. To avoid that cost, a set of sensors, or batch, can be deployed and each sensor is placed in a position $p_1, p_2, \ldots, p_o$ so that it can detect the incident within a range $r_1, r_2, \ldots, r_o$. For each measurement $y$, the resulting measurement is either $m_y = (p_y - r_y..p_y + r_y)$, if the incident is within its vicinities, or its negation intersected with $I$, otherwise. The intersection of all measurements gives an updated incident interval which may be a single interval or a disjunction of intervals.

Each measurement follows the model in fig. 2.1. Depending on the cost of the sensor and the measurement that it can take to check an incident interval, it is decided how many measurements, which kind of measurements, where to take measurements and how long before taking new measurements.

![Figure 2.1: Overview of Measurement Process](image)

The process of measurement can be composed into five stages: (a) waiting stage, where the sensor is stored for a given time until it is deployed; (b) deployment stage, where the sensor is deployed to a given position; (c) placement stage, where the sensor is placed at a given position; (d) detection stage, where the sensor detects or not an incident in its vicinities and broadcasts its information; (e) decaying stage, where the broadcast information is bound to the time of detection and loses accuracy over time; (f) prediction stage, where the decayed information is used to predict the progression of the incident and review which sensors to deploy.

This model was designed in a way that tries to identify the most relevant aspects and problems by looking at each one of the stages. For instance, the following sections of this chapter describe charac-
teristics that were considered in this thesis, and section 6.2.2 present others that might be considered in future work. However, other challenges may be brought by analyzing and most likely extending this model.

2.2.1 Waiting Time

Given the spontaneous nature of the incidents, permanent surveillance would be wasteful, in particular when its cost is greater than the incident itself. Conversely, during an incident, not every area is critical enough to be constantly monitored. The presence of sensors may influence the incident itself in a way that can difficult its containment.

Example 2.1. Assuming a situation where a sensor can be stationed and replaced at a given place in the sea and take a measurement per minute, for the entire sea to be thoroughly watched, it would be required a number of sensors that is proportional to the size of the sea and frequency of replacements. The cost of this system would be the cost of all deployments plus the cost to maintain all sensors. If an adversarial submarine crosses that sea once a week, or $10080$ minutes, during $200$ minutes, then only approximately $2\%$ of the overall cost would be justifiable.

An alternative to this approach would be to have a cheaper yet imprecise sensor that would detect if a submarine is in the sea, regardless of its position within it. Following that event, a more granular investigation could be made to check if the submarine is within any sensitive area. Nevertheless, even in this case, where the sensitive areas represent a fraction of the sea, the constant measurement of each kilometer would bring more costs than benefits. To find out, for instance, whether the submarine is in $(0..1)$ is not useful when the probability of being there is lower than in any other kilometer.

For that reason, every deployment is scheduled to be taken after a waiting time, when the measurement is most useful.

2.2.2 Waiting Cost & Deployment Cost

In the short term, there are two factors to take into account when taking measurements: how much a batch of measurements cost, and how many decisions will be made based on those measurements. As said in section 2.2, cheaper decisions can be taken over more expensive ones if backed with enough measurements. That happens because a measurement shows if an incident is happening in a sensitive area or not. The cost that is generated by other measurement-based decisions while not deploying sensors is called a waiting cost.

Example 2.2. To ensure safety at all times, the system could always trigger the red alert. It would not be necessary to take any measurement and simplify the entire decision process. However, the system
would cost 1000 dollars per second even if the submarine was not in the red area. To prevent that, if the system had a set of measurements that could rule out the possibility of the submarine being in the red area, then it could trigger a cheaper alert, like the yellow alert, or none.

However, since each measurement also has a cost called deployment cost, the cost of deploying an entire batch may surpass the waiting cost. The incident may have already been caught to be either inside or outside a sensitive area and sending more sensors would be useless. And, if that is not the case, then there may exist cheaper batches that can generate a lower or equal waiting cost.

**Example 2.3.** Checking that the submarine is not in a red area might prevent the usage of the red alert. However, if the sum of all the probes is more expensive than the cost of the red alert, then its deployment is considered a liability. There is a possibility that the submarine is not in the red area, and the red alert is triggered nonetheless.

### 2.2.3 Detection Range

Depending on the sensor mechanism and the surrounding environment, each type of sensor has its detection range for a given position. For instance, the sensor may be a complex electromagnetic device, that may be decalibrated. Another example can be found in Global Positioning System (GPS) communication, where buildings reduce the accuracy of GPS readings [3].

Assuming that the placement was precise, which may not happen outside this thesis, the sensor generates an interval whose center is the sensor position while in the detection stage. To narrow down this interval and provide a more granular analysis, output produced from sensors at different locations can be compared and intersected.

**Example 2.4.** Taking the example of the submarine, the red alert is not used if the system is certain that the submarine is not in (0..45). To achieve that, it must send sensors that cover the red area. Having sensors with a degree of uncertainty of 5 kilometers, it only requires a probe A at kilometer 40 to cover [35..45], and a probe B at kilometer 45 to cover [40..50].

This divides the sea in 5 regions: (a) [0..35], if neither A nor B detects the submarine; (b) [35..40), if only A detects the submarine; (c) [40..45], if both A and B detects the submarine; (d) [45..50], if only B detects the submarine; (e) [50..100], same as (a). That way, the red alert is only triggered when (c) happens.

### 2.2.4 Decaying Speed

Given the progressive nature of an incident, the measurements should follow their growth. This is made by extending the measurement interval at a speed called decaying speed. While this keeps the
measurement from becoming obsolete, the precision is reduced over time until it becomes large enough to be rendered useless. Because of that, sensors may be placed in the same location at different times.

**Example 2.5.** Having a moving submarine with a speed of 1 kilometer per minute, if a probe detects its whereabouts in \([30..40]\), then the knowledge space should be expanded at the same rate as the submarine. After 5 minutes, the value would be \([25..45]\) if no decision is made until then. After 60 minutes, the value would be \([0..100]\), which is the same as the sea area, becoming useless.

### 2.2.5 Prediction Distribution

The movement of incidents is continuous; it can hold still, and also can move sideways, but cannot instantaneously jump from one position to another. For that reason, having an incident confined within a set of measurements, it is not assumed that it always has the same probability of being in each position.

<table>
<thead>
<tr>
<th>Table 2.1: Probability Mass Function (PMF) of incident (x) when time (t = 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P(x \in (30..31)))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2.2: PMF of incident (x) when time (t = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P(x \in (29..30)))</td>
</tr>
<tr>
<td>(P(x \in (30..31)))</td>
</tr>
<tr>
<td>(P(x \in (31..32)))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2.3: PMF of incident (x) when time (t = 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P(x \in (28..29)))</td>
</tr>
<tr>
<td>(P(x \in (29..30)))</td>
</tr>
<tr>
<td>(P(x \in (30..31)))</td>
</tr>
<tr>
<td>(P(x \in (31..32)))</td>
</tr>
<tr>
<td>(P(x \in (32..33)))</td>
</tr>
</tbody>
</table>

In the example above, an incident was confined in a single unit interval. So the probability of being in that same space is 100%, as seen in table 2.1. Over time, without taking any other measurement, the incident interval grows and the probability in the previous single unit interval is equally distributed among the adjacent unit intervals. In table 2.2, the probabilities are the same as a single probability was split, but in table 2.3, where that does not happen, the probabilities are different.

Although this example was an idealistic one, since real-life incidents don’t always have the same probability of standing still and moving to each side, it’s the only one that is considered in the following chapters. It also presents a prediction distribution, where the incident is seen to be in different places and different probabilities.

This has influence over which measurements to take. In two similar situations, with the same taken measurements, two distinct sets of sensors may be deployed.

**Example 2.6.** We can have two similar submarine incidents that are not happening at the same time, where measurements indicate they are either in between \([30..45]\) or \([65..80]\). In the first scenario, there
is a much higher probability of the submarine being in the red zone (40..45) than in between (65..70) and, in this case, the system may send a sensor at kilometer 45 to cover the red zone. In the second scenario, let’s assume the submarine is much more interested in being in between (65..70) and there is a much higher probability of being there than in the red zone. In this particular case, the system may deploy a sensor at kilometer 65.

2.3 Summary

After presenting the motivation of starting this thesis, we have given some examples where sensors were used on-demand to optimize costs: an Adversarial Submarine that was navigating in the middle of sensitive naval assets; a Building Fire burning through rooms and corridors; a Building Fire with tumors in between vital organs.

A generalized model of Measurement-based Decisions was made to cover all examples, with some aspects such as: Waiting Time, where measurements may be needed to be taken at a given time. Waiting Cost & Deployment Cost, where both waiting and deploying a sensor implies a cost. Detection Range, where measurements have an inherent observational error. Decaying Speed, where measurements lose accuracy over time. Prediction Distribution, where the PMF of an incident happening throughout a given area is nonuniform.
3 Architecture

Contents

3.1 Possible Approaches ......................................................... 19
3.2 SenseAct ................................................................. 22
3.3 Summary ................................................................. 38
This chapter presents Possible Approaches that conceptualize agents that successfully support Measurement-based Decisions. Each one of them presents its advantages and disadvantages, and they are ordered so that the following approach solves any disadvantage that the previous approach has. Apart from them, a final approach is presented called SenseAct.

### 3.1 Possible Approaches

#### 3.1.1 Optimal Agent

This first approach is, in theory, the best one: the incident would be contained with the minimal amount of measurements and, to make that happen, an “Oracle” would, at the most appropriate time, recommend the best places to measure the incident.

![Figure 3.1: Optimal Agent - Component Diagram](image)

**Example 3.1.** For instance, when a submarine is known to be close to the red area, the “Oracle” can recommend the deployment of a cheap sensor to prove that the submarine is still outside the red area. Otherwise, if the “Oracle” knows that the submarine is already on the red area, it does not recommend anything.

In practice, this approach defeats the purpose of having measurements. If the “Oracle” already knows how the incident changes, then there is no need to get input from the environment. Nonetheless, it serves as a baseline of SenseAct performance and quality tests.

#### 3.1.2 Greedy Reflex Agent

Having the ideal architecture covered, we can now proceed with the most simple one. In a greedy approach, the incident would be always pinpointed through constant measurements. For that to happen, initially, when there were no measurements to locate the incident, the entire environment would be sprinkled with sensors, which would pinpoint the incident. Afterward, to keep the knowing location of the incident as accurately as possible, a batch of sensors would be deployed in its vicinity, to frequently check the incident location. In cases where the cost of deploying a batch is more expensive than the waiting cost itself, the batch is not sent.
Example 3.2. On the first minute, where it is known that the submarine is between \([0..100]\), a batch of sensors is sent so that each kilometer is covered by a combination of measurements. In this particular case, let's say that a subset of sensors has detected the submarine, so its whereabouts are restricted to the interval \((1..2)\). In the following minutes, the interval where the submarine is located, as well as its adjacent intervals, are covered by measurements. That way, the location of the submarine is always restricted to a single interval unit.

However, batches are always being sent to keep the incident location pinpointed, even when it is not useful. In most cases, when it is certain that the incident will be in the same sensitive area, it is useless to send any batch and expend any deployment cost in the process. Unless the deployment cost is virtually none, this approach is wasteful in cases where the incident does not often change its position to other sensitive areas.

3.1.3 Lazy Reflex Agent

To optimize the deployment cost, we could opt for a more lazy approach, where the measurements are only taken when it is not certain in which sensitive area the incident is occurring. When that happens, the incident would be pinpointed, like in the Greedy Reflex Agent, but within the boundaries where the incident might be located.

Example 3.3. Following example 3.2, we have a submarine in between \((1..2)\). Assuming a situation where the submarine does not move anymore, the measurements will decay over time. On one hand, a Greedy Reflex Agent would keep sending batches indefinitely, spending deployment cost when the waiting cost is always 0. On the other hand, a Lazy Reflex Agent would instead keep the measurements
decaying until it reaches the interval $[0..51]$. When that happens, a batch would be sent to measure every kilometer unit between 0 and 51. From this batch, the agent would know that the submarine has remained in between $(1..2)$. This process would be repeated over time.

As seen in example 3.3, every batch that is sent to the environment is intended to measure every interval unit where the incident might be occurring, only to find out that the incident has remained in the same place. In other words, the Lazy Reflex Agent does not predict or plan when and where the sensors will be deployed, making it use more sensors than the optimal amount.

### 3.1.4 Monte Carlo Tree Search (MCTS) Learning Agent

To add prediction and planning capabilities to an agent, we need to take into account their computation complexity. Not having enough heuristics to know exactly how the incident behaves, we have conceived a Cost Table where, for every time and interval where the incident might be located, we have a set of distinct batches that were previously found to be cost-effective into determining the incident location with a predetermined number of sensors. This would already give us a complexity of $O(\Delta t \times p^2 d)$, where $\Delta t$ is the period of time in which the plan takes place, $p$ is the number of space unit the environment has, and $d$ is the number of dimensions that the environment has. Taking into account a search algorithm for each time and interval, we would most likely have an agent that is unable to react to incident changes in real-time if the computation is not made elsewhere.

For that reason, we would need to divide the computation into two parts: a part where most of the computation is made, either in the background or ahead of time; and a part that interacts with the environment with real-time constraints. This would give us an architecture that would be an instance of a "general learning agent", which was conceptualized and illustrated in Artificial Intelligence: A Modern Approach (AIMA) [4, p. 54-57]. In short, and following AIMA logic, we would have a performance element which interacts with the environment in real-time, a learning element that would acquire and build knowledge for the performance element, a critic which evaluates actions made by the performance element and notifies the learning element about possible enhancements, and a problem generator that proposes new actions for the performance element to make for learning purposes. To consider both planning and prediction analysis, we would divide the learning element so that the planning part, which would require no input in real-time, would be made beforehand and the prediction part, which depends on input in real-time, would be made in parallel with the performance element.

An immediate approach that follows the previously mentioned model would be a MCTS Learning Agent, where a variation of the MCTS [5] with heuristics would cover the part of the learning element that is processed ahead of time, as well as the problem generator and the critic. The following adaptations would be made to use the MCTS in this case. Firstly, the Cost Table would be represented by the Monte Carlo tree, where each level of depth would represent the time, each node from the level would
be an interval, and each directed edge would have a batch. Secondly, the altered MCTS would not have a final win, loss or draw result. Instead, the tree would have a predefined maximum depth, which would be the extension of the plan, so that either one of the MCTS steps that navigate from the parent node to the child node - selection, expansion or extension - would end when they reach the maximum depth. The outcome would then be evaluated according to the accumulated sum of the waiting costs and deployment costs, and its result would be backpropagated to the child nodes. Finally, there would be the requirement of filling the Cost Table before ending the MCTS. This would involve checking if every node above the maximum depth would have at least one child node and every node below the minimum depth would have a parent node.

Figure 3.4: MCTS Learning Agent - Component Diagram

However, even if the heuristics are optimized, this modified MCTS scores the best-evaluated paths as a whole and accumulates their rewards, keeping alternative paths from having a possibly better overall score. SenseAct presents a more granular approach, where the search is made at each incident interval.

3.2 SenseAct

An alternative approach to MCTS has been conceived, called SenseAct, with decisions that depend on its immediate consequences, instead of the path in which it was made or the ending outcome. To do
that, we have decided to conceive an algorithm that makes time-backward simulations through dynamic programming. In other words, we have a final moment $t_{\text{Final}}$, where we consider each interval to have zero cost and, by successively going back a time unit to the moment $t_{-n} = t_{\text{Final}} - n$, we search cost-effective batches for each interval at $t_{-n}$ based on the cost of the intervals at $t_{-(n+1)}$.

SenseAct presents a similar architecture as MCTS Learning Agent. Following the same “general learning agent” model from AIMA, this agent does not rely on a MCTS. Instead, it uses a GA, which mimics the behavior of natural selection through reproduction at cellular level [4, p. 126-129], to find the most fitting batches for a given interval and time. The GA evaluation method is prepared for this particular case, where several costs are calculated by accessing values from the cost table that were previously generated. The GA instances are managed by the learning element to reduce computation resources.

![SenseAct Learning Agent - Component Diagram](image)

This architecture is explained in the following sections.

### 3.2.1 Performance Element

The performance element follows the model defined in the Measurement-based Decisions in real-time. To do that without any computation delay, it consults a few predefined tables with their respective columns, which are explained throughout this section: Sensor Table: detection range and deployment
In a round-robin approach, it repeats a set of instructions, while keeping track of the number of iterations it has already done using a counter. As seen in fig. 3.6, the cycle is divided in three groups of instructions:

**Deploy batch:** It starts by looking into already taken measurements, which are stored in the measurement table, and fold them with intersections to get a disjunction of intervals. Each interval of this disjunction is used as a key, in conjunction with the counter, to select the cheapest batch from the cost table. The deployment cost of each sensor is extracted from the sensor table.

**Receive measurements:** After their deployment, the sensors reply if they detect an incident within their vicinity or not. In this thesis, the deployment process is instantaneous, so the measurements are obtained right after being requested, but it could be obtained after some time. If the incident is detected, the respective reply is interpreted as an interval whose size is twice the detection range seen in the sensor table. Otherwise, it is interpreted as a negation of the same interval. In both cases, the information is stored in the measurement table.

**Wait for a time unit:** Finally, the performance element waits a given time unit until repeating the iteration. While waiting, it recalculates the disjunction of intervals from the recently updated measurement table and intersects it with all area intervals from the area table. If the disjunction intersects...
with any one of the area intervals, it uses the intersected area intervals to get their respective waiting costs from the area table; only the most expensive is considered. After getting the waiting cost, all intervals from the measurement table are extended a decaying speed from their original size.

3.2.1.A Sensor Table

This is simply a table that contains all information regarding sensors. Each row has information of a specific kind of sensors, such as the detection range and the deployment cost. This used to get the interval that is covered by a given deployed sensor, as stated in the previous section. It is also used to obtain the cost of a sensor when SenseAct is evaluating its usage in a specific context (section 3.2.2.D).

Table 3.1: Adversarial Submarine Sensor Table

<table>
<thead>
<tr>
<th></th>
<th>Detection Range (km)</th>
<th>Deployment Cost ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cheap Sensor</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Expensive Sensor</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

3.2.1.B Area Table

In the area table, each row has information about a given area, with the corresponding real interval and waiting cost. This is exemplified in table 3.2.

Table 3.2: Adversarial Submarine Area Table

<table>
<thead>
<tr>
<th></th>
<th>Area Interval (km)</th>
<th>Waiting Cost ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Alert</td>
<td>(40..45)</td>
<td>1000</td>
</tr>
<tr>
<td>Yellow Alert</td>
<td>(45..70)</td>
<td>50</td>
</tr>
</tbody>
</table>

3.2.1.C Measurement Table

The knowledge state of the incident is represented by the measurement table. It is where measurements taken by sensors are accumulated, and it is used to restrict the position that the incident might adopt. Without those restrictions, SenseAct must assume the incident can be anywhere. An example of a measurement table can be seen in table 3.3.

Table 3.3: Measurement Table $t = 1\text{min}$

<table>
<thead>
<tr>
<th></th>
<th>Measurement Interval (km)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[37..38]</td>
</tr>
<tr>
<td></td>
<td>(42..46)</td>
</tr>
</tbody>
</table>

The measurement table is initialized empty, without any measurement to restrict the incident interval. Since this would imply measurement-based decisions with the highest waiting cost, a batch is deployed.
to collect evidence that validates the containment of the incident in a given area with a cheaper waiting cost.

Every time a measurement arrives, it is intersected with the measurement table. If the sensor of the measurement detects an incident in its surroundings, then its position and its absolute uncertainty compose, respectively, the center and the radius of the interval. Otherwise, the same interval will be negated, generating a disjunction of two intervals (table 3.4).

<table>
<thead>
<tr>
<th>Placement Point (km)</th>
<th>Detection Range (km)</th>
<th>Has Detected Incident?</th>
<th>Measurement Interval (km)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>2</td>
<td>No</td>
<td>(−∞..38) or (42.. + ∞)</td>
</tr>
<tr>
<td>41</td>
<td>5</td>
<td>Yes</td>
<td>[36..46]</td>
</tr>
<tr>
<td>42</td>
<td>5</td>
<td>Yes</td>
<td>[37..47]</td>
</tr>
</tbody>
</table>

As each time unit passes by, given the movement of the incident, the measurement table has to expand all its intervals at the same rate as the max speed of the incident (section 2.2.4). Eventually, the intervals from the measurement table are unable to decay anymore, becoming useless. If no more sensors are going to be deployed, then the measurement table will discard the useless intervals and return to its initial value.

3.2.1.D Cost Table

The cost table is where the planning towards a given space is stored. Knowing the extension of its plan, the SenseAct knows which batches should send given the evidence it has for a time frame. For that, the cost table is indexed not only with the interval where the incident might be but also with the time already passed since the starting moment.

<table>
<thead>
<tr>
<th>Planning Time Frame (min)</th>
<th>Incident Interval (km)</th>
<th>Cheapest Batch</th>
<th>Cheapest Collateral Cost</th>
<th>First Batch</th>
<th>Second Batch</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>35</td>
<td>(40..41)</td>
<td>[..., Yes, No, No, Yes, ...]</td>
<td>56.2434234</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>35</td>
<td>(40..42)</td>
<td>[..., Yes, No, Yes, Yes, ...]</td>
<td>78.4372422</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>35</td>
<td>(40..43)</td>
<td>[..., No, Yes, No, Yes, ...]</td>
<td>56.75752934</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Before deploying SenseAct, the Ahead-of-time (AOT) Learning Element executes several GA instances to search for batches to be deployed at a given time and a specific set of evidence. Without input, it is assumed that the probability distribution of an incident being at each unit area is uniform. For that reason, each cost table key matches several different batches.

The selection of the cheapest batch is made while the Performance Element builds the measurement table. When that happens, the Just-in-time (JIT) Learning Element uses the disjunction from the
measurement table to build the probability distribution into a PMF. This data structure is then used by the Critic to recalculate the cost of deploying the batches.

### 3.2.2 Critic: GA Evaluation Heuristics

The collateral cost of every batch taken at a given time and circumstance is calculated by an evaluation algorithm. This method relies on the Performance Element tables since they will have already predefined values, and it is used by the JIT Learning Element to select the best-evaluated batch. In a limited capacity, as explained in section 3.2.3.C, it is also used by the Problem Generator to get the fitness of its population of batches.

![Figure 3.7: SenseAct Critic - Activity Diagram](image)

As described in fig. 3.7, once they are received by the Critic, they are used to generate a set of intervals and weights, which are used to calculate the cost behind deploying them. The collateral cost is the sum of the three kinds of costs: Mean Waiting Cost (MWC), Mean Subsequent Collateral Cost (MSCC), and Batch Deployment Cost (BDC). These costs, as well as the Intervals and Weights, are explained throughout the following sections.

#### 3.2.2.A Intervals and Weights

The critic algorithm begins by receiving a batch and intersecting the measurements that could be obtained by them into disjunctions of intervals. Each intersection represents a potential situation in real-life,
where the incident is happening in an interval where can be detected by a part of the batch. The example 2.4 is an example where a given batch may return different measurements, depending on the positioning of the incident.

As these intersections have different relevances due to their size or the way the incident progresses, each interval has an associated weight. This weight is made by summing all intervals from the probability table that is contained in the intersection.

\[ P(x \in i) = \sum \{ \text{PMF}(y) \mid y \cap i \neq \emptyset \}, \]  

(3.1)

where \( i \) and \( y \) are intervals, 
\( x \) is the incidents position, 
\( P(x \in i) \) is a probability of incident position \( x \) belonging to a given interval \( i \)

**Example 3.4.** Let’s consider example 2.4, where two measurements allowed SenseAct to consider 5 regions, grouped into 4 scenarios, for the reasons already presented in example 2.4: (a) or (e), (b), (c) and (d). Let’s also assume a probability table identical to table 3.6. For simplicity’s sake, the probability distribution between kilometer 34 and 54 is uniform to multiply the distribution constant with the size of the region. In any other example, the distribution may not be uniform and all values according to their region would have to be summed. The probability of having a measurement in a given interval would be the following: (a) \( 0.05 \times 1 = 0.05 \); (b) \( 0.05 \times 5 = 0.25 \); (c) \( 0.05 \times 5 = 0.25 \); (d) \( 0.05 \times 5 = 0.25 \); (e) \( 0.05 \times 4 = 0.20 \).

### 3.2.2.B MWC

With this probability, the overall cost of deploying a batch at a given moment is obtained through a divide to conquer approach, where the respective intervals of each possible measurement are intersected with all areas. For each one of them, the MWC is calculated as the most expensive waiting cost whose area interval is intersected with the disjunction of intervals multiplied by the sum of all interval probabilities associated with the respective measurement interval.
\[
\text{MWC}(M) = \sum_{D \in M} \left\{ \max_{i \in D} \{ \max \{ \text{areaTable}(y) \mid y \cap i \neq \emptyset \} \} \times \sum_{i \in D} P(x \in i) \right\},
\]
(3.2)

where \(i\) and \(y\) are intervals,

- \(x\) is the incidents position,
- \(M\) is a set of batch measurements,
- \(D\) is a disjunction of intervals,

\(\text{MWC}(M)\) is the MWC of a set of batch measurements \(M\)

### 3.2.2.C MSCC

There is another cost that needs to be taken into account, which is associated with the cost of waiting a given amount of time. For each possible measurement, a decayed interval is obtained after a given amount of time. Depending on the time left for the cost table to render useless, the decayed interval can be used to get the cheapest cost from its respective row in the cost table and multiply it with the probability associated with the measurement. Let's call this value the MSCC.

\[
\text{MSCC}(M, \Delta t) = \sum_{D \in M} \left\{ \sum_{i \in D} \left\{ \min \{ \text{costTable}(i \pm e_s, \Delta t + e_t) \} \times P(x \in i) \right\} \right\},
\]
(3.3)

where \(i\) is an interval,

- \(x\) is the incidents position,
- \(e_s\) is a space unit,
- \(t\) is a point in time,
- \(e_t\) is a time unit,
- \(M\) is a set of batch measurements,
- \(D\) is a disjunction of intervals,

\(\text{MSCC}(M, \Delta t)\) is the MSCC of a set of batch measurements \(M\) for a planning time frame \(\Delta t\)

### 3.2.2.D BDC

Finally, the BDC is obtained by identifying the type of sensors used in the batch and used it as a key to get the respective deployment cost from the sensor table.
3.2.3 Problem Generator: GA

Before any external input, SenseAct applies GAs under different incident intervals and planning time frame in search of cheaper batches. Using some batches from a previous search as feed, the first generation of batches is generated and, throughout successive cycles, the best-evaluated batches are selected and mutated into the next population until a stopping condition is satisfied.

This component depends on the Critic as the evaluation method to obtain the collateral cost of each batch, which serves as their fitness value.

![Figure 3.8: SenseAct Problem Generator - Activity Diagram](image)

3.2.3.A First Generation

This algorithm involves creating chromosomes whose bits represent the decision of deploying a given kind of sensor at a given location. This generation depends on the probability of flipping a bit to true, called generation probability. For each kind of sensor, SenseAct has a generation probability $P(g)$, and, for each bit that is about to define, the probability is compared to a random value between 0 and 1.

The number of possible locations to put the sensor in a given incident interval is previously defined
as a parameter \#Points. In the end, the method generates a chromosome with the size of the number of possible locations times the number of sensor types. This process is repeated until the first population is generated. Its size is also predetermined as a parameter.

### Algorithm 3.1: First Generation Strategy

\begin{verbatim}
begin
  #Points \in \mathbb{N}_0 \land SeqP(g) \land \mathcal{P}_{Elite} \in \mathcal{P}(B)
  // Define population of empty batches
  \mathcal{P}_{Empty} \in \mathcal{P}(B) \leftarrow \{ [\bot] \times \#Points \times \#SeqP(g) \}
  // Define population of randomly generated batches
  \mathcal{P}_{Random} \in \mathcal{P}(B) \leftarrow \emptyset
  repeat
    c \leftarrow []
    foreach \mathcal{P}(g) \in SeqP(g) do
      repeat
        c \leftarrow \text{rand} \leq \mathcal{P}(g)
        until repeating \#Points times
      p_{Random} \leftarrow \{ c \}
    until repeating \#\mathcal{P}_{All} - \#\mathcal{P}_{Elite} times
  // Merge populations
  return \mathcal{P}_{Elite} \parallel \mathcal{P}_{Empty} \parallel \mathcal{P}_{Random}
end
\end{verbatim}

### 3.2.3.B Next Generations

Following the evaluation of the current population, each chromosome is bound to a fitness value, which defines how they are ordered. The fitter they are, the closer they are to become the first element of the population and higher the probability of being selected. After being selected, they only pass through a mutation process before being added to a new population.

Each parent chromosome is picked up through eq. (3.4). When applied to a random value between 0 and 1, the resulting value ranges from 0 to \(\infty\), with a probability distribution of \(\frac{1}{x^b}\), where \(r\) is the random value and \(b\) is the logarithm base. Assuming a random value with an equal probability of becoming any value between 0 and 1, the probability of this method of returning \(n\) is the double of returning \(n + 1\).
\[ n(r, b, P) = \min \{ \text{round}(- \log_b r) + 1, \#P \}, \quad (3.4) \]

where \( r \) is a random number between 0 and 1,

\( b \) is a logarithm base,

\( P \) is the chromosome population,

\( n(r, b, P) \) is a value to access the \( n^{th} \) fittest element of population \( P \).

The selected chromosome is mapped to flip each gene. The flipping process is bound to a condition of flipping probability \( P(\text{flip to } b \text{ given } C) \), which is defined by eq. (3.5). The \( P(\text{flip}) \) is defined by settings, while the rest of the equation depends on the number of genes equal to 1. Afterward, the mutated chromosome is inserted into the child population.

\[
P(\text{flip to } b \text{ given } C) = \begin{cases} 
P(\text{flip}) \times \frac{1 + \sum x}{\#C}, & b = T \\
0 & \text{otherwise}
\end{cases} \quad (3.5)
\]

where \( P(\text{flip}) \) is the settings-defined flipping probability,

\( x \) is a gene boolean value,

\( C \) is a chromosome,

\( P(\text{flip to } b \text{ given } C) \) is the probability of flipping a bit to \( b \) given a chromosome \( C \).

This process is repeated until the child population reaches the same size as its parent population.

**Algorithm 3.2: Next Generations Strategy**

\begin{algorithm}
begin
\[ p_{\text{Parent}} \land \#p_{\text{Child}} \in \mathbb{N}_0 \land P(\text{flip}) \in \mathbb{R}^+ \land 0 \leq P(\text{flip}) \leq 1 \land \text{base} \in \mathbb{R}^+ \]
\[ p_{\text{Child}} \leftarrow (p_n)_{n=1}^{\#p_{\text{Child}}} \]
repeat
\[ C \leftarrow p_{\text{h(}rand_1^{\text{base,}p_{\text{Parent}}})} \]
// Update chromosome by randomly flipping bits
foreach gene \in C do
\[ \text{gene} \leftarrow \text{rand}_1 \leq P(\text{flip to } \neg \text{gene} \text{ given } C) \]
// Insert updated chromosome into child population
\[ p_{\text{Child}} \leftarrow \{ C \} \]
until \#p_{\text{Child}} = \#p_{\text{Parent}} \]
return \( p_{\text{Child}} \)
\end{algorithm}
3.2.3.C Evaluation using Critic

After generating a population either by the first time or not, each chromosome is evaluated by the Critic. The best-evaluated chromosomes are the ones with the smallest collateral cost. The chromosomes are then ordered by their evaluation, which is useful when the GA is selecting a chromosome from the population using eq. (3.4).

As previously said in section 3.2.2, the usage of the Critic in the GA is limited. This is due to the lack of input. Without an input to predict what is the progression of the incident, the GA is forced to assume the PMF is uniform.

3.2.3.D Stopping Condition

The GA repeats the process of generating and evaluating a population until it checks that the best-evaluated chromosome has not been replaced for a given number of consecutive iterations. When the GA produce a chromosome with zero cost, it becomes useless. When that happens, the search ends immediately.

Algorithm 3.3: Stopping Condition Strategy

\[
\text{begin } \quad \text{Fit} \in \mathbb{P}([0, +\infty)) \land \max\text{Counter} \in \mathbb{N}_0 \land \text{counter} \in \mathbb{N}_0 \land \text{fit}_{\text{Min}} \in \mathbb{R}_0^+ \\
// \text{ Increment counter if fittest chromosome from population is the best overall} \\
\text{fit}_{\text{Next}} \leftarrow \min\{\text{Fit}\} \\
\text{counter} \leftarrow \begin{cases} 
0, & \text{if fit}_{\text{Next}} < \text{fit}_{\text{Min}} \\
\text{counter} + 1, & \text{otherwise} 
\end{cases} \\
\text{fit}_{\text{Min}} \leftarrow \text{fit}_{\text{Next}} \\
// \text{ End GA if counter reaches a predefined value or best fitness reaches 0} \\
\text{return } \text{counter} \geq \max\text{Counter} \lor \text{fit}_{\text{Min}} = 0
\]

3.2.4 AOT Learning Element: Dynamic Programming Heuristics

The AOT Learning Element is responsible for filling the cost table with batches from GA instances without having to satisfy real-time constraints and performance requirements. Even so, it also relies on a few heuristics to avoid, reduce and reuse GA-related computation, respectively: calculate Well-Known Collateral Costs, aggregate Useful Interval Groups and aggregate Endpoint Groups.

Since it does not have any customized information about how a specific incident behaves, it stores more than one batch from each GA instance into the cost table. Later, during an incident, each group of batches will be periodically reevaluated by the other part of the Learning Element: the JIT Learning Element.
3.2.4.A Well-Known Collateral Costs

Before searching for batches, a few trivial situations, where the measurements indicate that the incident is inside an area, are ruled out. In those cases, it is not necessary to have a single sensor at any location in the next time unit. For that reason, the collateral cost depends only on the MWC and MSCC; the equation of both costs are also simpler versions of eq. (3.2) and eq. (3.3), respectively.

\[ C_{Cost}(i, \Delta t) = \max \{ area_{Table}(y) \mid y \cap i \neq \emptyset \} + \min \{ cost_{Table}(i \pm e_s, \Delta t + e_t) \} \]

where \( i \) and \( y \) are intervals,
\( e_s \) is a space unit,
\( t \) is a point in time,
\( e_t \) is a time unit,

\( C_{Cost}(i, t) \) is a Collateral Cost in a given interval \( i \) for a planning time frame \( \Delta t \)

As seen in eq. (3.6), the collateral cost is the sum of the MWC and the MSCC when SenseAct decides not to send a single batch. Being certain that an incident is within a given area, \( P(x \in i) \) from
eq. (3.2) and eq. (3.3) are equal to 1, becoming redundant in eq. (3.6).

\[
R((x,y), \Delta t, d, (p_{Min}, p_{Max})) = (d_{Min}, d_{Max})
\]

(3.7)

where

\[
d_{Min} = \begin{cases} 
  x + (\Delta t - 1) * d, & \text{if } x > p_{Min} \\
  x, & \text{otherwise}
\end{cases}
\]

\[
d_{Max} = \begin{cases} 
  y - (\Delta t - 1) * d, & \text{if } y < p_{Max} \\
  y, & \text{otherwise}
\end{cases}
\]

There are three circumstances where sending batches is useless: incidents in exact locations; incidents within unit intervals; incident within larger intervals in a reduced area, which is calculated through eq. (3.7). In the first two cases, SenseAct does not investigate further into unit intervals or smaller, unless the size of the unit interval is redefined in the meantime. In the latter, the incident does not surpass the area during the remainder of the planning time frame.

In algorithm 3.4, the eq. (3.6) is applied to all the above cases, and the results are stored into the Cost Table.

**Algorithm 3.4: Well-Known Collateral Costs Strategy**

begin
\[ M \equiv \mathbb{P}(\mathbb{R}) \cap (p_{Min}, p_{Max}) \in M \land \Delta t \in \mathbb{N} \land s, d \in (\mathbb{R}_0^+) \land d \mod s = 0 \]

foreach \[ ((a_{Min}, a_{Max}) \in areaTable) \]

// Get reduced area; reduced if endpoints are not absolute
\[
(d_{Min}, d_{Max}) \leftarrow R((a_{Min}, a_{Max}), \Delta t, d, (p_{Min}, p_{Max}))
\]

// Get intervals with zero length, unit length, or within reduced area
\[
L_0 \leftarrow \{(l..r) | a_{Min} \leq l = r \leq a_{Max} \land l \mod s = 0 \}
\]
\[
L_1 \leftarrow \{(l..r) | a_{Min} \leq l < l + s = r \leq a_{Max} \land l \mod s = 0 \}
\]
\[
L_{>1} \leftarrow \{(l..r) | d_{Min} \leq l < l + s < r \leq d_{Max} \land \{l, r\} \mod s = \{0,0\} \}
\]

// Store Collateral Cost in the intervals above and \( \Delta t \)
foreach \[ i \in L_0 \cup L_1 \cup L_{>1} \]

\[
costTable(\Delta t, i) \leftarrow \{[\cdot, C_{Cost(i, \Delta t)}]\}
\]

3.2.4.B Useful Interval Groups

Once all incident intervals with Well-Known Collateral Costs are identified, the remaining incident interval is grouped by batch. By knowing beforehand which batch is better used to investigate an incident, the usage GA can be reduced.

During the search of a batch over an incident interval, the GA always tend to cover some of the area intervals, ignoring at least one of them. Taking that into account, useful intervals are obtained by negating an area over an incident interval. By limiting the searches for useful intervals only, the resulted
batches on the respective incident intervals are reused.

**Algorithm 3.5: Useful Interval Groups Strategy**

\[
\begin{align*}
\text{begin} \\
M \equiv \mathbb{P}(\mathbb{R}^n) \land (p_{\text{Min}}..p_{\text{Max}}) \in M \land \Delta t \in \mathbb{N}_0 \land s, d \in (\mathbb{R}_0^+) \land d \text{ mod } s = 0 \\
U_{\text{Groups}}(u) : M \rightarrow \mathbb{P}(M) \leftarrow \emptyset \\
\text{foreach} \ (l..r) \in \{ (l..r) | p_{\text{Min}} \leq l < l + s < r \leq p_{\text{Max}} \land (r - l) \text{ mod } s = 0 \} \ do \\
\quad \text{uCost} \leftarrow 0 \land \text{uZone} \leftarrow [] \\
\quad \text{foreach} \ [(a_{\text{Min}}..a_{\text{Max}}), \text{waitCost}] \in \text{areaTable} \ do \\
\quad \quad (d_{\text{Min}}..d_{\text{Max}}) \leftarrow R((a_{\text{Min}}..a_{\text{Max}}), \Delta t, d, (p_{\text{Min}}..p_{\text{Max}})) \\
\quad \quad \text{if } d_{\text{Min}} \leq l < d_{\text{Max}} \ then \\
\quad \quad \quad u_{\text{Min}} \leftarrow d_{\text{Max}} - s \land \text{uCost} \leftarrow \text{waitCost} \land \text{uZone} \leftarrow (d_{\text{Min}}..d_{\text{Max}}) \\
\quad \quad \quad \text{break} \\
\quad \text{foreach} \ [(a_{\text{Min}}..a_{\text{Max}}), \text{waitCost}] \in \text{areaTable} \ do \\
\quad \quad (d_{\text{Min}}..d_{\text{Max}}) \leftarrow R((a_{\text{Min}}..a_{\text{Max}}), \Delta t, d, (p_{\text{Min}}..p_{\text{Max}})) \\
\quad \quad \text{if } d_{\text{Min}} < l \leq d_{\text{Max}} \land (d_{\text{Min}}..d_{\text{Max}}) \neq \text{uZone} \ then \\
\quad \quad \quad u_{\text{Max}} \leftarrow d_{\text{Min}} + s \\
\quad \quad \quad U \leftarrow \\
\quad \quad \quad \quad \{ (l..u_{\text{Max}}) \}, \quad \text{if } u_{\text{Min}} \text{ is unassigned} \\
\quad \quad \quad \quad \{ (u_{\text{Min}}..u_{\text{Max}}) \}, \quad \text{if } \text{waitCost} = \text{uCost} \\
\quad \quad \quad \quad \{ (l..u_{\text{Max}}), (u_{\text{Min}}..r) \}, \quad \text{otherwise} \\
\quad \quad \quad \text{break} \\
\quad \text{if } U \text{ is unassigned} \ then \\
\quad \quad U \leftarrow \{ (l..r) \} \\
\quad \text{foreach} \ u \in U \ do \\
\quad \quad U_{\text{Groups}}(u) \leftarrow \{ (l..r) \} \\
\text{return } U_{\text{Groups}} \\
\end{align*}
\]

3.2.4.C Endpoint Groups

After getting a set of useful intervals \( U \), the GA computation is further optimized by grouping the useful intervals by their endpoints. When the batches on a given useful interval are computed, the same batches are used as baselines on a useful interval similar to the previous one, but with a small increment in one of the endpoints.

3.2.4.D GA-based Batches

With both Useful Interval Groups and Endpoint Groups identified, the GAs are managed in a way that for each endpoint group \( E \), each incident interval \( e \) is used as a place to send batches in GA. Each interval is extended, to place sensors where they can still measure the incident: if the detection range is larger, the extension is wider. Also, populations from previous GAs, which are called \( p_{\text{Elite}} \) in algorithm 3.7,
Algorithm 3.6: Aggregation by Similarity Strategy

begin
$M \equiv P(R^{n}) \wedge U \in P(M)$

// Count interval endpoints
$E(position, type): R^{n} \times B \rightarrow N_{0} \leftarrow 0$

foreach $(l, r) \in U$ do
$E(l, \perp) \leftarrow 1 \wedge E(r, T) \leftarrow 1$

// Group intervals by most common endpoint
$EGroups(position, type): R^{n} \times B \rightarrow P(M) \leftarrow \emptyset$

while $U \neq \emptyset$ do

$e \leftarrow any\{x \mid E(x) = \max E\}$

foreach $(l, r) \in U$ do

if $e \in \{[l, \perp], [r, T]\}$ then

$EGroups(e) \leftarrow \{(l, r)\} \wedge E(l, \perp) \leftarrow 1 \wedge E(r, T) \leftarrow 1 \wedge U \leftarrow \emptyset$


return $EGroups$


have their chromosomes extended with padding, to match the extended incident interval.

Every time a new population is extracted from the Problem Generator, SenseAct gets the Useful Interval Groups associated with $e$ from the $UGroups$. For each interval $u$ in the Useful Interval Groups, each chromosome on the population has additional padding, to match the length of $u$. Additionally, the padded population $p_{elite}$ is evaluated by the Critic to get the collateral cost, so that it can be added to the Cost Table with the $p_{elite}$.

Algorithm 3.7: GA-based Batches Strategy

begin
$M \equiv P(R^{n}) \wedge \Delta t \in N_{0} \wedge s \in P(R^{+}) \wedge UGroups : M \rightarrow P(M) \wedge E \in P(M)$

foreach $e \in E$ do

// Prepare and execute Problem Generator
$d \leftarrow e$, extended with largest detection range \& $p_{elite} \leftarrow \emptyset$ with padding
$p \leftarrow GA$ within $d$ using $p_{elite}$

foreach $u \in UGroups(e)$ do

// Prepare and execute Critic; store results in Cost Table
$p_{elite} \leftarrow p$ with padding
$costTable(\Delta t, u) \leftarrow \{p_{elite}, Fitness of p_{elite}\}$

3.2.5 JIT Learning Element: Prediction Heuristics

This learning element completes the cost table provided by the AOT Learning Element with real-time input. This component is composed of a prediction phase, where it uses a PMF, following by a re-
evaluation phase, where the Critic is used against values from the cost table, this time with the PMF to weight the cost of each measurement combination. The best-evaluated batches go to their respective cheapest batch field in the cost table.

3.2.5. A PMF

In the prediction phase, the PMF is used to predict the progression of the incident. Being a data structure that completes the measurement table with probabilities for each interval unit, it is updated through a probability propagation method, when waiting for measurements, and a probability cropping method, when the received measurements restrict the incident interval.

<table>
<thead>
<tr>
<th>Position (km)</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(34..35), (35..36), ..., (53..54)</td>
<td>0.05 (each)</td>
</tr>
<tr>
<td>Otherwise</td>
<td>0.0</td>
</tr>
</tbody>
</table>

3.3 Summary

Having the overall problem identified and structured in section 2.2, we have proceeded with several agents that were conceptualized to solve them.

We started by naming a few alternatives that are considered to support section 2.2, but in more limited ways than SenseAct. An idealistic agent was defined with capabilities to predict where the incident was taking place and advised which measurements to take, however, we concluded its implementation was not doable in real life. Then, we have listed some simple reflex agents, which were used as straw men to be compared with SenseAct in the chapter 5. They do not use any planning or searching algorithm and all their components can be used in real-time, however, they do not optimize the overall cost because they scatter sensors, some of them too expensive to be useful, without knowing when or where to send them. A more complex learning agent that uses a Monte Carlo Algorithm was also defined. Despite being able to plan where to send probes over a given planning time frame, the resulting plans are path-dependent and, thus, not able to be adapted through prediction analysis.

Finally, we described SenseAct in five components: Performance Element, which deploys batches in real-time based on previous measurements. Critic, which estimates the collateral cost of a batch over a planning time frame. Problem Generator, which searches cost-effective batches through a best-effort GA. AOT Learning Element, which identifies common problems to execute fewer GA instances. JIT Learning Element, which reevaluate batches in real-time with a naive predictive analysis.
This chapter explains how SenseAct was implemented. In section 4.1, this thesis shows why Python was used as the only language to write code for SenseAct, as well as its limitations. In section 4.2, some notes are made about the fact that SenseAct does not rely on any framework in particular. It also shows the system environment where it was developed. In section 4.3, a few aspects of SenseAct development were mentioned, like program profiling, code quality, and unit testing.

4.1 Programming Language: Python

SenseAct is a library that works in Python [6]. Since the focus of this thesis was to create a proof-of-concept framework, the language-related performance was not critical for its evaluation, namely multi-threading. Conversely, the expressiveness of the algorithms was important to determine if the algorithm itself was effective and efficient without the risk of implementing a different one while translating the pseudocode into code. Also, the popularity of the language [7–10] was taken into account, since it means that more people can understand and further develop SenseAct.

4.1.1 Parallelism Limitation: Global Interpreter Lock (GIL)

There are situations in SenseAct where the parallelism would be useful. For instance, in each time frame, the AOT Learning Element looks into each Endpoint Groups without having to share resources between them. By parallelizing this part, the computation of Problem Generator would be split between the cores.

However, Python does not support multi-core, since its GIL involves a mutex that locks Python objects [11]. An implementation of SenseAct with multi-processing was made, but it was not passing the tests by the time this thesis was written.

4.2 Dependencies

SenseAct does not use any particular library that is not a Python built-in module. On one hand, although any GA-related framework would provide a range of standardized algorithms and statistic functionality, they would limit the chromosome format and method interface. On the other hand, their usage would make SenseAct dependant of any bug, change or support they could have, make or end respectively.

SenseAct was implemented on Windows 10, using the official main implementation of Python interpreter for program execution. Since it was not necessary to communicate with other different processes, there was neither any other component that was used nor any tool to integrate SenseAct with any external application.
4.2.1 Alternative Dependency: Distributed Evolutionary Algorithms in Python (DEAP)

An example of a framework that could have been used for the GAs is DEAP [12]. While it was used in an initial stage of development, its framework would limit SenseAct-specific implementations of the GA. For instance, at that time, some of the Useful Interval Groups features from AOT Learning Element were in the Problem Generator instead, more specifically in the main function of the GA. Also, the tool was not usable without invoking some functions to configure its toolbox, and would make the AOT Learning Element less expressive and transparent, and harder to debug. For those reasons, DEAP was removed from SenseAct.

4.3 Other Settings

Although without any further implication besides ensuring that the SenseAct implementation is reliable, some development stages were covered. By using an Integrated Development Environment (IDE) called PyCharm [13], SenseAct was implemented using plugins like the profiler [14], for program profiling and SonarLint [15], to check code quality and consequently to avoid language-specific pitfalls and bugs. Besides, Python provides a testing tool called unittest [16]. With this tool, it was possible to certify that all SenseAct components were well implemented.
Contents

5.1 Evaluation Environment ........................................ 45
5.2 AOT Learning Element Performance .......................... 45
5.3 Performance Element Real-Time Constraints ............... 47
5.4 SenseAct versus Straw Men ...................................... 48
5.5 Summary ............................................................. 49
5.1 Evaluation Environment

The evaluation process of the AOT Learning Element Performance was made on a set of five Ubuntu 18.04.2 Long Term Support (LTS) [17] within a cluster. Each one of them has an Intel® Xeon® Processor E5506 [18], with eight cores at 2.13 GHz, and a Random-Access Memory (RAM) of 40 GB. With this, our computation-intensive tests can be made on a steady pace, to avoid variance caused by factors other than the testing process and the evaluation environment.

Since the remaining evaluation criteria are not dependant on performance, their results are taken from a single Windows 10 Home [19] running on an Intel® Core™ i5-7200U Processor [20], with two cores at 2.50 GHz, and a RAM of 8 GB.

Each one of these environments has a Python interpreter, like in the development environment. All tests were made through Python scripts calling SenseAct modules.

5.2 AOT Learning Element Performance

This test evaluates the overall performance of SenseAct, as well as its applicability. While the Performance Element must indeed satisfy real-time constraints, most of the computation resources is allocated to the AOT Learning Element. Being the element where the Cost Table is built, it is also the most susceptible to lose performance over a more realistic setting, considering the complexity involved in both the Critic and Problem Generator.

A clocking method was placed to measure the duration of each iteration in the AOT Learning Element main function. As seen in section 3.2.4, the higher the number of iterations this learning agent can make to build a single Cost Table, the larger this table will be, and, consequently, the longer is the planning time frame, in which SenseAct will know how to deploy its batches. Every time an iteration is made, a planning time frame is considered and its performance is logged to create the fig. 5.1, which is seen below.

This chart comprises the results of twenty executions of the AOT Learning Element; four for each cluster machine and scenario (5 machines times 4 scenarios). In common, all scenarios follow the submarine example. Three variables were taken into account for each scenario: the deployment cost, or “Sensor”; a cheap “Yellow” area cost; and a more expensive “Red” area cost. In the x-axis, the planning time frame is placed in intervals of 5 minutes, while, in the y-axis, the performance time is in seconds. The purpose of this chart is to see how long the cost table is built for each planning time frame.

As seen in fig. 5.1, the performance time linearly grows as the planning time frame time gets higher and its respective portion of the Cost Table is inserted. This is justified because the complexity of the genetic algorithm search grows with the complexity of the Cost Table itself. Initially, the cost table is filled only for the 0 planning time frame, where all incident intervals are associated with empty batches, with a
batch cost of 0. Over time, as the AOT Learning Element considers higher planning time frames and the GA-based Batches fills the cost table with batches that are not empty, with their respective cost defined by Critic, the Cost Table gets a larger diversity of batches and costs. Moreover, the cost of an incident intervals at a planning time frame propagates to its surrounding incident intervals at a higher planning time frame. Nonetheless, the rising speed decreases at two points: between 25 and 30 time unit, and between 55 and 60. This coincides with the planning time frames where the reduced areas without any waiting cost involved cannot be reduced any more because they are empty. In those cases, the Cost Table still gets more complex, only on a smaller scale, because the incident intervals stop to propagate its cost to incident intervals with zero cost.
5.3 Performance Element Real-Time Constraints

For this evaluation, where the SenseAct performance is tested in real-time, we simulate a situation where each agent has a routine that repeats every minute of execution time. In each routine, each agent uses its algorithm to receive any on-going measurement, so that it can make decisions with the most updated information. In this case, the decision implementation is instantaneous, so the only real-time constraint is to have an algorithm that is faster than a minute. The decay phase, where each measurement is decayed and eventually removed, has not been considered in this metric, because that is executed in separate.

Like in section 5.2, the Performance Element has a clocking method at the beginning and at the end of the component method. This test has also the same four settings fig. 5.1 has, however, it also simultaneously considers four agents: Greedy Reflex Agent, Lazy Reflex Agent, Optimal Agent and SenseAct. In this case, for each setting, we have a chart where we compare the performance of each agent in each routine, in 2000 simulations of 100 routines, which are 100 for each execution in section 5.2 (100 times 20), in a total of 200000 samples (2000 simulations times 100 routines).

![Figure 5.2: Performance Element Performance - Box and Whisker Chart](image)

As predicted, none of the agents violates any real-time constraint. As seen in fig. 5.2, every sample is faster than half a decisecond (0.05 ds). Every statistical metric present in the charts indicates that most samples reach values lower than a millisecond. Since the means are also lower than a millisecond, the outlier samples are not representative. Also, they present such values most likely because the evaluation environment was executing other processes in parallel.
5.4 SenseAct versus Straw Men

The last evaluation sees how different the quality of the decisions made by SenseAct and this thesis straw men are. While the agents and the settings are the same as in section 5.3, we are expecting results that are visibly different from each other, unlike what happened in fig. 5.2. For that reason, the evaluation is also divided into three parts, each one organized in the same settings and agents as fig. 5.2. The first two parts show the metrics that were used to evaluate the quality of the agents, while the last one joins the previous values for an overall evaluation.

In common, the Optimal Agent is used as baseline, so it should adopt the lowest values. This evaluation also use the same 200000 samples as section 5.3.

In particular, there is this scenario "Sensor: 500, Yellow: 500, Red: 1000", where the cost of the sensor is almost the same as the most expensive waiting cost, and its results would differ the most. This scenario is made to see how the agents behave when they need to judiciously consider if a single measurement should be taken or not, otherwise its cost will be greater than the one caused by uncertainty.

5.4.1 Waiting Cost

The first group of charts measures the accumulated waiting cost at each end of the routine, i.e. in each execution time in minutes. It is expected for the SenseAct to have a slightly higher waiting cost over time in comparison with the rest of the agents because it should balance between covering sensitive areas to lower the waiting cost, and sparing sensors to have a lower deployment cost. In this case, both the greedy and the lazy approaches, which were built to minimize waiting cost, should have similar values as the baseline.

In the upper-left chart of fig. 5.3, we can see that all agents have similar quartiles and medians. However, SenseAct quartile tend to get higher than the remaining ones when the gap between the "Sensor", "Yellow" and "Red" costs narrow. The SenseAct mean is also the highest mean in all scenarios.

5.4.2 Deployment Cost

This group is similar with the previous one, in section 5.4.3, but it measures accumulated deployment cost instead of waiting cost. In theory, it is in this part of the evaluation where SenseAct should have better results: while the greedy and the lazy approaches favours the usage of sensors, SenseAct sends what it considers cost-effective.

In short, as seen in fig. 5.4, all statistical values from SenseAct are lower than the ones from the greedy and the lazy approaches. In scenario "Sensor: 500, Yellow: 500, Red: 1000", SenseAct values are even lower than the baseline, but that is not significant, since the Optimal Agent we made does not
avoid sending sensors that are more expensive than the cost of waiting. If that was the case, then the baseline would have the lowest values.

5.4.3 Overall Cost

This last group presents the sum of the charts from section 5.4.3 and section 5.4.2 as the overall quality evaluation of SenseAct. It is where we can see what happens when SenseAct tries to minimize both the waiting cost and the deployment cost.

Through fig. 5.5, we can see that, in the overall, SenseAct has better results than the alternative approaches. Despite having the most valued mean in “Sensor: 10, Yellow: 50, Red: 1000”, its quartiles and medians are all lower than the greedy and the lazy approach. In fact, some of the samples reach values near the equivalent ones from the optimal approach.

5.5 Summary

After implementing SenseAct, as described in chapter 4, we have relied on a cluster to get samples under environments prepared for computation-intensive yet steady processes. With them, we have evaluated SenseAct under the following criteria: AOT Learning Element Performance, where the AOT Learning Element is clocked while building cost tables on a sample of settings to check its performance; Performance Element Real-Time Constraints, where the Performance Element is verified to match a few
real-time constraints between receiving measurements and deploying new sensors; SenseAct versus Straw Men, where some of the overall costs taken from SenseAct are compared with the straw men from Architecture.
Figure 5.5: Performance Element Overall Cost - Box and Whisker Chart
6 Discussion

Contents

6.1 Result Interpretations ............................................. 55
6.2 Future Work ...................................................... 55
6.3 Summary .......................................................... 58
6.1 Result Interpretations

By looking at the Evaluation chapter, we have seen that, in general, SenseAct performs better than simple reflex agents such as the Greedy Reflex Agent and the Lazy Reflex Agent. However, some of its means have reached the highest value in their respective categories, which suggests that although the bulk of the sample was performing better than their alternative counterparts, some samples from SenseAct were behaving much worse. In part, this is because the Cost Table was built with few batches in each row, without using any previous data from the environment in which it should be applied. For the Cost Table to be reliable without any input or external-based heuristic, more batches should have been generated. Nonetheless, other improvements should be addressed, not only to produce better results from already-established settings, but also to enhance SenseAct, or at least its architecture and background, to tackle and solve new challenges and, eventually, be applied into Motivation Examples, which may be larger, more generalized and in larger number by then.

6.2 Future Work

The following improvements have been identified so that they can be used in future work.

6.2.1 Algorithmic Optimizations

6.2.1.A Other GAs

The Problem Generator considers only a single combination of GA methods. This thesis was elaborated as a proof of concept, to verify if the SenseAct approach applied to real-life situations. In future work, other GA methods could be applied and compared to this thesis, not only to enhance SenseAct in general, but also to tune it in specific situations, or groups of situations.

6.2.1.B Other Dynamic Programming Heuristics

The AOT Learning Element considers few heuristics that, although reduce the usage of GA, could be more elaborate. For instance, the Well-Known Collateral Costs does not consider situations where the deployment of batches is more costly than the costs they are trying to optimize.

6.2.1.C Predictive analytics in learning elements

The AOT Learning Element uses the Critic with a uniform PMF. Also, the JIT Learning Element has a rudimentary method to predict the progression of the incident. It evenly distributes the probability value of the PMF to adjacent probabilities, without considering heuristics that might change how they
are distributed over time. A more accurate solution would be to identify how many kinds of incidents exist and how each one of them behaves and progresses through previously-made research, and use that information to fill and possibly update the PMF while using the AOT Learning Element. After filling the Cost Table, the JIT Learning Element can use a . . .

6.2.2 Further generalization of Measurement-based Decisions

6.2.2.A Deployment Time

Sensors may be sent from a given location. Having some mean to be transported to the destination position, it will take a given deployment time $\Delta t_{\text{Deploy}}$ depending on the distance between the initial location $p_I$ and the final location $p_F$ and the speed of transportation $s_T$. This would follow the eq. (6.1).

$$\Delta t_{\text{Deploy}}(p_I, p_F, s_T) = \frac{p_F - p_I}{s_T},$$ (6.1)

where $p_I$ is the initial position,

$p_F$ is the final position,

$s_T$ is the speed of transportation,

$\Delta t_{\text{Deploy}}$ is the deployment time

For this to be applied in SenseAct, the JIT Learning Element would predict the progression of the incident, as defined in section 6.2.1.C, and notify the Performance Element that it would access values from the Cost Table associated to the predicted incidents at a planning time frame ahead of the current one. That way, the Performance Element would dispatch sensors that will arrive at the desired location at the same time as the remaining future batch.

To take advantage of the entire Cost Table in all the planning time frames it considers, the prediction would be made even before the occurrence of the incident.

6.2.2.B Deployment Fault & Placement Fault

Sensor deployment may be delayed, or even halted. For instance, the mechanism that transports the sensor may have some malfunction or the environment is adverse and full of obstacles. Moreover, this may also depend on the trajectory of the sensor. Although this does not affect the reliability of a measurement value, the lack of measurement at a given moment may increase the waiting cost.
6.2.2.C Placement Fault

The causes mentioned on section 6.2.2.B are also applied in this case. A problem with the transportation or on the environment may also affect the positioning of the sensor, changing the destination position and, therefore, the measurement. Although this problem is detected through some tracking mechanism like a GPS receiver, it could be also predicted and considered by SenseAct while generating the Cost Table.

6.2.2.D Decaying Fault

Usually, the Decaying Speed is the estimated maximum speed of an incident, however, this value could be lower than the real one. An immediate solution for this would be to make an increase in the Decaying Speed, however, this may not be applicable in situations where the Cost Table is already built and its reconstruction would take more time than required. A less accurate yet faster algorithm could be made to adjust the Cost Table with the updated Decaying Speed. An alternative for this could be to

6.2.2.E Prediction Fault

In a more complicated version of the Adversarial Submarine example, the submarine is capable of bypassing prediction analysis, either by being aware of its movements or by a security breach. A possible approach would be to predict how the submarine and incidents, in general, try to bypass prediction analysis.

6.2.2.F Detection Range with PMF

Given physical and electromagnetic phenomena, most sensors have a higher chance of misdetection over the edges of their detection range than in their center. For that to be pondered on by the Critic, each type of sensor could have a PMF over their detection range. When the Intervals and Weights are being calculated, there would be several identical intervals with different probabilities, where the sum is 1. This probability would be then multiplied by eq. (3.1).

6.2.2.G Dynamic sensitive areas

This thesis follows the example of the Adversarial Submarine, where the sensitive areas would not change over time. However, in the Building Fire, for instance, we have rooms that are more urgent to be checked only because they have people in it. In other words, the cost of waiting changes over the number of people in each room, and not in the room itself. From this situation, three scenarios can be modeled: a first one, where the number of people in each room is unknown, and the waiting cost is distributed among the rooms that are higher probability of having people; a second one, where the number of people in
each room is unknown, but there are surveillance mechanisms, like the ones described in section 2.1.2, that can discover the waiting cost; and a third one, where the number of people in each room is known, and the sensitive area and respective waiting cost are frequently updated.

The first case is already considered by SenseAct, although the waiting cost should be previously defined by an external tool, the second and the third cases present challenges that could be addressed in future work. For instance, there could be a PMF for the sensitive areas, in both cases, to predict how the sensitive areas are going to change. In the second case, in particular, the measurements would also have the ability to delimit the sensitive areas, and that would take a distinct yet similar way of approaching this as the one already defined by SenseAct. Both approaches could also overlap, for cost optimization and possibly identifying.

6.2.2.H Byzantine Fault Tolerance in Measurement-based Decisions

Sensors may lie, either by external manipulation or internal malfunction. This Byzantine Fault problem can be solved by simply replicating each one of the sensors for localized quorum consensus. However, it involves two problems: 1. the deployment cost is multiplied by the number of Byzantine Faults SenseAct can tolerate in each position; 2. sensors can be more easily manipulated in certain positions or when are closer to other sensors. Alternatives to replication could be considered to reduce the deployment cost, like mixing sensors with a smaller detection range with others that are either cheaper or cover a larger detection range.

6.3 Summary

In this last chapter before concluding this thesis, we’ve interpreted the results from chapter 5, justifying its values, in which ways they were acceptable and how they could be enhanced through future work.

Under the settings already established in this thesis, the following solutions have been mentioned: Other GAs, to improve SenseAct in general and to tune it for in particular cases; Other Dynamic Programming Heuristics, to optimize the performance of SenseAct; Predictive analytics in learning elements, for context-aware evaluation of batches.

Although not solved by SenseAct, these characteristics were also identified and modeled under the context of Measurement-based Decisions: Deployment Time, where sensors may take time to be deployed; Deployment Fault & Placement Fault, where sensors may not be deployed at the stipulated time; Placement Fault, where sensors may not be deployed in the right place; Decaying Fault, where the estimated Decaying Speed may be lower than the real one; Prediction Fault, where the incident progression may not be predicted in the most adequate way. Detection Range with PMF, where sensors may have less probability of detecting an incident near the range limit; Dynamic sensitive areas, where sensitive
areas may also be dynamic; They can react to incident progression, and vice-versa; Byzantine Fault Tolerance in Measurement-based Decisions, where wrong measurements may be taken by damaged or lying sensors.
Conclusion
In this thesis, we have established a model to cover a range of situations where the degree of uncertainty over the knowledge of an environment is associated with a given cost. This model covers the process of taking measurements to control the uncertainty involved. With this, various characteristics and challenges were identified and solved by conceiving an agent capable of deciding in real-time which measurements to take to optimize the overall cost. As seen through performance and quality evaluations, it successfully presents better results than its counterparts, needing, however, to generate data in advance to work.

Since the evaluation was focused on few simple cases, and considering the complexity of building a Cost Table, SenseAct only works as a proof of concept. However, in future work, the current architecture will be enhanced with prediction analysis, as well as alternative GA and heuristics for a faster and more context-aware agent. Furthermore, the model and, consequently, the architecture, will be generalized to include more complex use cases.
Bibliography


