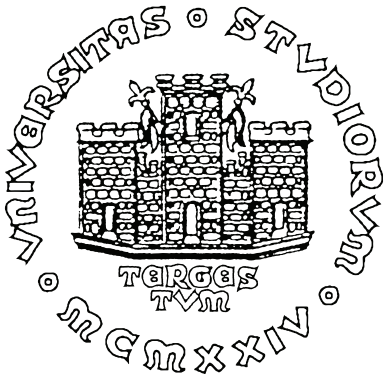


Università degli studi di Trieste

Dipartimento di Matematica e Geoscienze

Master's Degree in
Data Science and Scientific Computing

Optimizing fault search in power grid outages through Reinforcement Learning

Candidate:

Angela Carraro

Supervisor:

Antonio Celani, ICTP

Co-Supervisors:

Andrea Zancola, AcegasApsAmga

Luca Bortolussi, UniTS

Emanuele Panizon, ICTP

Academic year 2020/2021

A Matteo

Sommario

Riassunto in italiano.

Abstract

Contents

Acronyms

AAA AcegasApsAmga S.p.A.. 2, 25, 43

AWS Amazon Web Services. 38, 64

BFS Breadth-First Search. 49, 50

DFS Depth-First Search. 50

DTP Decision-Theoretic Planning. 12

GIS Geographic Information System platform. 25

MDP Markov Decision Process. 12, 13, 16, 17, 36

ML Machine Learning. 11

NPG Natural policy Gradient algorithm. 53–55, 57–61

PG (ordinary) Policy Gradient algorithm. 53–56, 58

POMDP Partially Observable Markov Decision Process. 16–18, 36, 38, 44

RL Reinforcement Learning. 11, 12, 15, 16, 21

Chapter 1

Introduction

Understandably, the restoration after a fault is an essential task in the operation of power systems. The *restoration process* returns the system back to normal operation after any combination of system components have been lost as a result of a fault. Restoration has traditionally been performed by qualified technicians, assisted by guidelines developed by utilities using knowledge gained through experience. But now, in the era of computers and big data, it is natural to try to find a way to model this problem computationally and to solve it using more advanced techniques. Over the course of this thesis, we will both model algorithmically the current technique used by the technicians and find a new method that can exploit the information of the environment that we have available.

In this chapter, we will analyze the power grid of the city of Trieste, with all its components, and how the faults that happen are handled. Specifically, we will focus on faults occurring on the medium voltage power grid. In fact, this is what we are going to model in subsequent chapters. To this day, the company managing the grid relies on specialized technicians who, using their experience, try to restore the electricity to all users as quickly as possible. We will see how they operate, and we will understand the problems that may arise.

1.1 Power grid elements

The power grid of Trieste is handled by AcegasApsAmga S.p.A. (AAA), an Italian company based in Trieste and subject to the direction and coordination of Hera S.p.A., also called Hera Group, which is a multiutility company based in Bologna, Italy. Hera operates in the sale and distribution of gas, water, energy, and waste disposal in some Italian northeastern provinces. It is also responsible for the infrastructures, on which it operates a technical and administrative management. AcegasApsAmga does the same, but is restricted to some municipalities of Veneto, Friuli-Venezia Giulia, and in the Balkans.

As part of the electricity supply chain, which is the sequence of production steps that start from the raw material and arrive at the finished product, AAA also deals with the transmission and the distribution of electricity. The *transmission* consists of the transport of electricity on the national power grid in high – or very high — voltage, to deliver it in the regional distribution networks to local distributors to which end users are connected. The *distribution* consists of transporting electricity on local networks for delivery to end customers, and in those activities related to all accessory operations for the connection of end users. The company is responsible for ensuring the safety and continuity of supply, providing for the management and maintenance of the entire electricity network and the systems connected to it. In particular, the company must also handle and resolve any blackouts deriving from faults that may occur on the power line, something called *restoration process*.

We will now describe Trieste's power grid, mainly the one in medium voltage — since it is the one we are interested in, and all its elements. The electricity produced from different primary sources of energy is transmitted in high or very high voltage to the *primary substations*, an electrical system that has the function of transforming the high voltage input energy into medium voltage energy. Then, the electricity is transmitted in medium voltage to the *secondary substations*, an electrical system for transforming electrical energy from medium voltage to low voltage, and is then distributed to the users. Actually, some users like hospitals, police or fire stations, big supermarkets or malls can receive the power directly in

medium voltage, and then use their transformers to convert it to low voltage.

Specifically, a substation is a complex of conductors, equipment, and machines designed to transform the voltage of the electrical lines. After being transported through the cables, the electricity arrives in the substation at the *(circuit) breakers*, which are electrical switches that allow to open or close a circuit, respectively stopping or allowing the power to flow. Then the power flows through a *busbar* (or two, depending on the substation, connected through a *conjunctor*), which is a metallic strip, or bar, used to carry and distribute the current to multiple circuits. Afterwards, it passes through a *transformer*, which decreases its voltage. The cables that connects breakers, busbars and transformers among them are called *connections*. At this point, if it is a primary substation, the electricity flows through another busbar, and finally exits the substation in medium voltage through a breaker. If it is a secondary substation, the electricity in medium voltage exits directly through the breaker to arrive at another secondary substation. There is also electricity in low voltage that exits the substation, which arrives at the final users, but since we have no interest in these elements, we will not discuss them. A *single-line diagram* is a symbolic schematic of the components of a three-phase electric system, which uses only one line to represent all three phases. In Figure 1.1 we can see an example of a single-line diagram of a primary substation, with a description of the main elements that we just mentioned.

Instead, an *electrical diagram* is a schematic that represents the electrical connections between the various elements of the power grid — the primary and secondary substations and the electrical cables that connect them – using standardized symbolic representations. The presentation of the interconnections in these diagrams does not necessarily correspond to the real arrangements, and, most importantly, the lengths of the cables are not to scale. They are mainly used to know the power supply sequence of the secondary substations in the *standard setup*, which means when everything is working fine and all the faulty elements are fixed. In Figure 1.2, we can see an electrical diagram of three lines of Trieste's power grid, indicated with different colors. The big rectangles at the top are the two primary substations, while the small squares are the secondary substa-

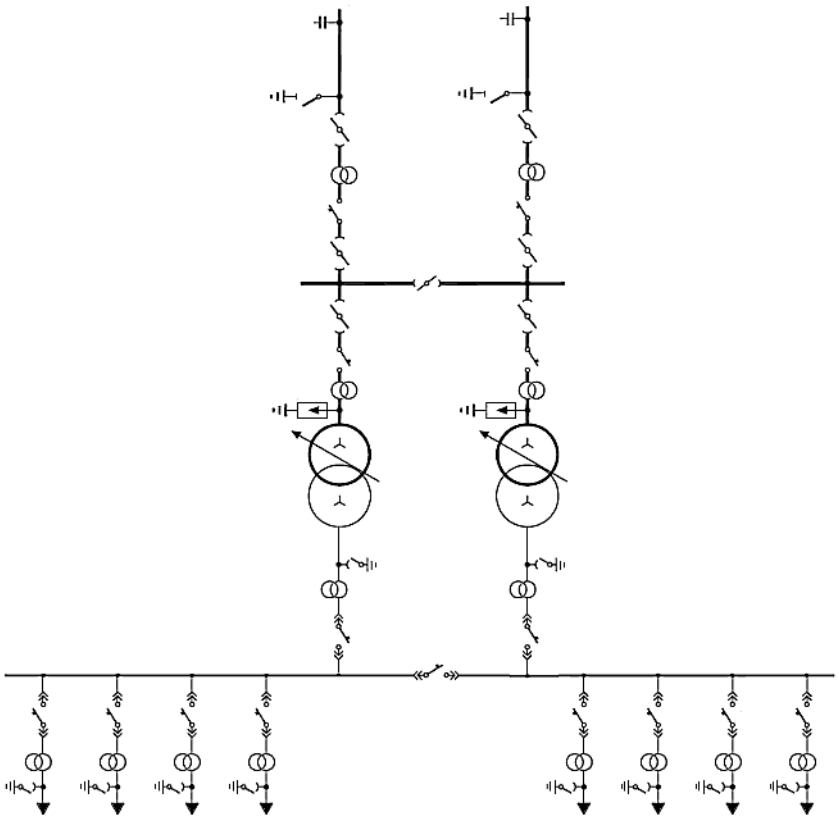


Figure 1.1: Example of a single-line diagram of a primary substation, with a description of the main elements (adapted from Wikipedia).

tions. The yellow circles specify which are the remotely controlled substations — note that all the primary substations are. In a *remotely controlled substation*, the breakers are motorized in order to be able to be operated remotely. This happens from the *remote control room*, which is also responsible for monitoring the entire network. The black circles connected to some secondary substations indicate the *points of delivery* (pods) in medium voltage, like the ones to hospitals, stations, malls, or supermarkets. The hyphen that can be seen attached to some substations indicates that the breaker of that cable in that substation is open in the standard setup, which means that the electricity doesn't flow in that substation

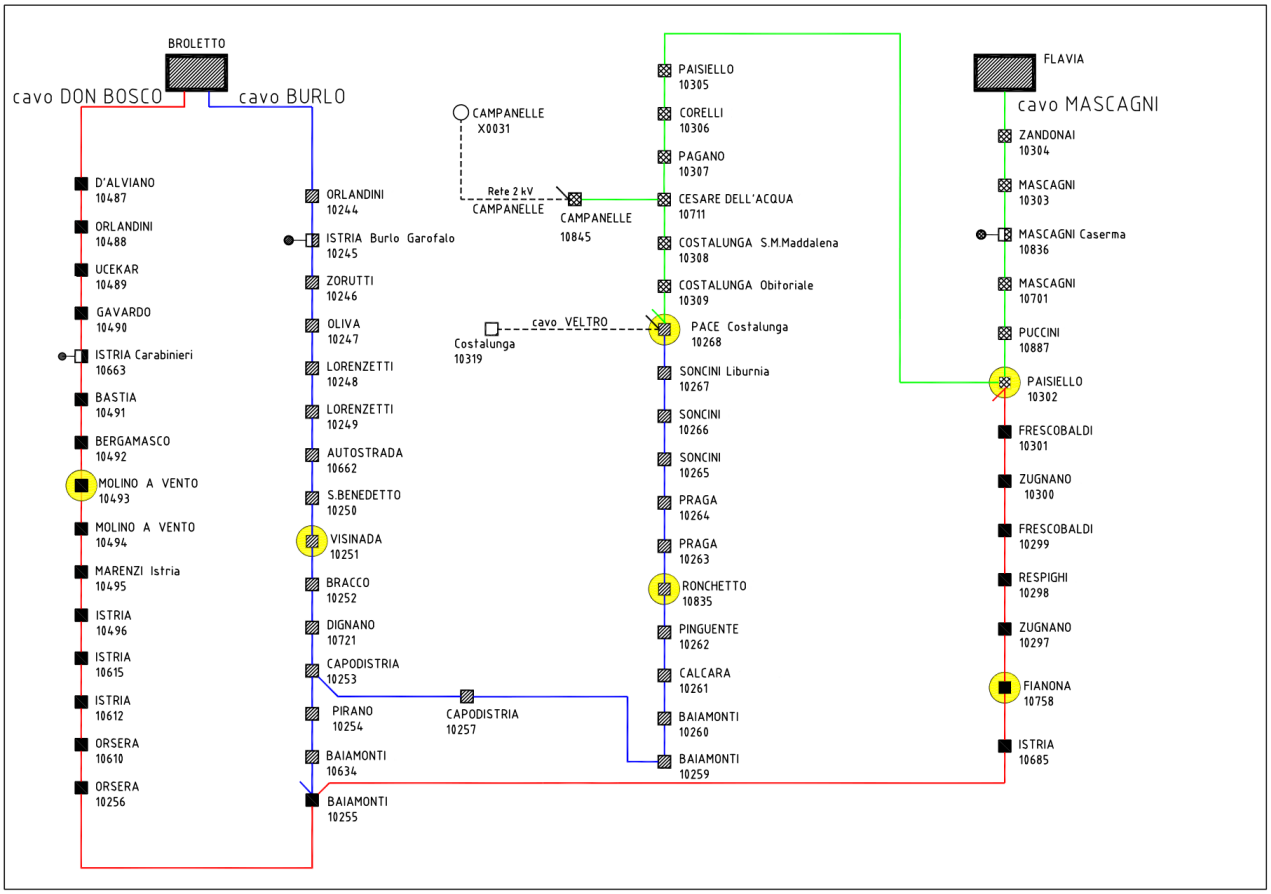


Figure 1.2: Electrical diagram of three power lines of Trieste's power grid, indicated with different colors. The big rectangles at the top are the two primary substations, while the small squares are the secondary substations. The yellow circles specify which are the remotely controlled substations (all the primary substations are). The hyphen which can be seen attached to some substations indicates that the breaker of that cable in that substation is open in the standard setup. The black circles next to some secondary substations indicate the points of delivery in medium voltage.

through that cable. This derives from the fact that we have a *meshed electrical grid*, which means that the power grid allows for alternative interconnection paths between any two substations, and therefore allows powering the same substation from different power grid cables, ensuring greater continuity of service. Thanks to this redundancy, after a fault, the users are reconnected in very few hours rerouting the power through other cables, while the technicians restore the damaged line, which remains deactivated and thus can be safely fixed. After everything is repaired, the power flow is brought back to the standard setup. It's worth highlighting that, however, the cost of meshed electrical grids limits their application to medium voltage transmission and distribution networks.

1.2 When a failure happens

A fault can happen for various reasons and in all possible elements of the power grid. If it is one of the elements of a substation that breaks, often it is because it is flawed, and when the technician visits that substations, they can notice it right away, and usually they just have to replace that element. If the fault happens on an electrical cable, it is usually because it is no longer perfectly isolated, for example, because the wire's insulation breaks down — caused maybe by some workers that cut the cable while doing road works, or by some rats that chew it, or simply by the deterioration of the insulating material. This causes a short circuit since the cable is connected to the ground and allows the charge to flow through it.

What happens after a fault on the medium voltage power line has occurred? The fault creates a short circuit, so a fault current propagates backward to the primary substation, and causes the breaker in the latter to open, disconnecting all the secondary substations that were powered by that substation through this specific cable line. A *fault current* is a current different from the normal one, as it has different properties; first of all, it has a much higher amperage, even around 1000 amperes (the normal one is around 50 amperes). A substation can see if there has been a fault through the *fault detectors*: they are devices that analyze the current, and if it is a fault current, then they know that there has been a fault

in some substation after this one.

The opening of the breaker and the disconnection of the substations trigger the alarm in the remote control room; at that point, the operators immediately alert the technicians, and the restoration process begins. First of all, the technicians ask the remote control room to check which fault detector detected the fault, in order to know in which segment of the power line the problem is. In this way, they are able to figure out the section of the line among two remotely controlled substations in which the fault happened. Then they proceed to remotely operate the breakers of the remotely controlled substations in order to restore the power to the substations outside this tract. For the substations after the fault, they use the power of another primary substation to power them, closing one of the breakers of the open cables: this is possible thanks to the meshed electrical grid. These operations are very fast: they take around 5 minutes in total. After this, they are left with only a small subset of non remotely controlled substations, which the technician has to visit in person to reconnect while searching for the fault. They will be called the initially disconnected substations in subsequent chapters. This manual restoration process is the part of the problem we will focus on, modeling it computationally and searching for a better algorithm to solve it.

1.3 The manual restoration process

To this day, the technique that the technicians follow to decide which substation to visit among the initially disconnected ones is *binary search*, which they call *bisection*. This means that they go in the substation that is in the middle (considering the number of disconnected substations), or almost in the middle, depending on the ease of access to the substation and maneuvering of the breakers (which could be old and break easily). It can happen, for example, that a substation is in a bank; accessing it at night could easily take several hours, because the technician has to wait for the security service. A substation could also be located in a fenced courtyard, have a faulty door lock, or be inaccessible due to adverse weather conditions.

Once in the substation, the technician has the possibility of finding out if the fault is before or after the substation in which they currently are. This is done with two different methods, and the one to use depends on many factors, among which the length of the electrical cable and whether it is day or night. The first one, the most used, is the *instrumental test*, which consists of opening a breaker in the substation to divide the cable into two segments, and attaching a detecting device to one of them. It has a needle for the voltage and a needle for the current, and it works by measuring how much current the cable draws when applying a voltage to it. If the tension needle moves, but the current needle remains on zero, then the cable is fine; while if the tension needle stays on zero, but the current needle goes to the full scale of the instrument (the maximum amplitude the system can represent), then on this cable there is a fault. In this way, this device can tell if the fault is before or after the current substation. However, this test is only possible if the length of the cable being checked is less than 2 km. If the cable is too long, the technician has to go to another substation of the line and open a breaker, so to reduce the length of the cable that is being tested, and then return to the substation to actually perform the test. This procedure is typically employed during the daytime, and if there are important users connected to the line, like hospitals, police stations, or firehouses.

The second method consists in simply proceeding by trial-and-error: once in a substation, the technician opens the breaker of the cable that exits the substation and goes into the next one, and asks the remote control room to close the breaker in the primary substation, repowering the line. If the cable doesn't short circuit again, this means that the fault is after the current substation, otherwise, it is before it. A couple of remarks on this method are in order. First, this test has to be done only on the cable that has experienced the failure, not on other ones, to avoid creating a blackout for users connected to an unbroken line. Second, the technician opens the breaker of the cable that exits the substation, and not the one of the cable that enters it, because, if the power does not fail — which means that the fault is after the current substation, this substation is already powered back on. Third, if instead the fault is before the current substation, closing the

breaker in the primary substation causes the power to fail again, which means that all the users on this entire line will be disconnected again and experience a second blackout. For this reason, this method is used only if the first one is not available, or if it is night, when there are much fewer active users, and is especially not used in presence of important users, like the ones mentioned before.

After having discovered if the fault is before or after the current substation, the technician can reconnect the segment of the electrical line that doesn't contain the fault. Thus, they can reconnect up to half the substations each time, so the time it takes to find the fault is in the order of the logarithm of the number of substations, and this is why the technicians use binary search. The fault is considered solved once all the users are reconnected again.

The technicians aim at reconnecting the users in the quickest way possible not only for the mere desire of offering a good service, but also for a more financial reason: each fault has an actual cost for the company. In fact, the company has to refund all the users that were disconnected. If the fault lasts more than an hour, the reimbursement that they have to pay is much higher, so it is important to act fast. Each user is refunded based on the time they have been disconnected. Thus, the cost of the fault is computed as the time each underlying user of each substation remains disconnected.

In the next chapters, we will model this problem and we will look for an algorithm that allows us to optimize the sequence of visited substations in order to minimize this cost. The algorithm has to decide in which substation the technician has to go after the one they are currently in, in order to solve the fault with the minimum cost. In fact, we will find an algorithm that can do better than just "going halfway", taking advantage of the information we have at our disposal.

Chapter 2

Reinforcement Learning

2.1 Introduction

Reinforcement Learning (RL) is a paradigm of Machine Learning (ML), along with others, like supervised learning and unsupervised learning [1]. It analyzes how it is possible to learn the best course of action in an environment, by maximizing some given numerical reward. The learner is not told which actions to take, but it rather empirically learns which are the ones that lead to a bigger reward. It is also possible that the current action influences not only the immediate reward, but also the next status of the environment, and, through that, all the other rewards to come. In this case, we speak of delayed reward, which, together with trial-and-error search, form the two distinctive features of RL.

In an RL problem we have a learning *agent*, the decision-maker, that interacts over time with the *environment* in which it is placed — which includes everything outside the agent — in order to achieve a *goal*, which is to maximize the total reward it receives over the long run. As we see in Figure 2.1, the agent can sense aspects of its environment through the *states* — which can also be seen as a representation of the environment itself, it can choose *actions* to influence the environment, and it receives a *reward* based on the outcome of its actions. We consider to be a reinforcement learning method any method that is well suited to

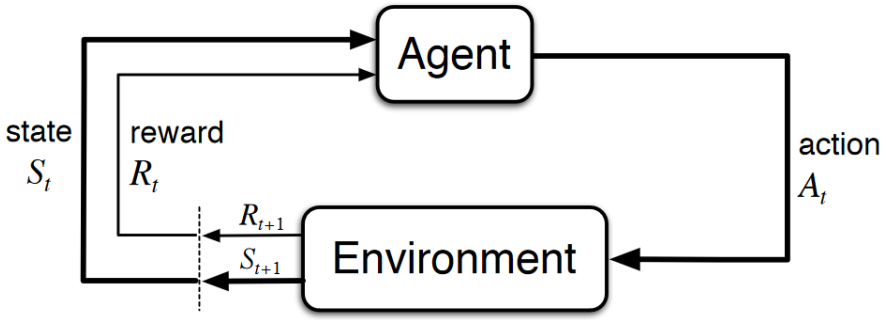


Figure 2.1: The agent-environment interaction [1].

solve problems framed in this way. Other elements of the RL system are a *policy*, a *value function*, and a *model* of the environment. A *policy* describes the behavior of the agent at a given time, using a mapping from states to actions to be taken in those states. A *value function* determines the value of a state as the total amount of reward an agent can expect to accumulate over the future, starting from that state. Instead, a *model* of the environment reproduces the behavior of the latter, and it is used for planning actions before actually experiencing them. In other words, when the model is provided with a state and action, it predicts with a certain probability the resultant next state and next reward.

2.2 Finite Markov Decision Processes

The treatment of this topic will closely follow the one presented in [1].

Markov Decision Processes, or MDPs, are used to formalize sequential decision-making, where actions influence the given rewards and the next states of the system, and through the latter future rewards can be affected as well. Thus, MDPs need to balance both immediate and delayed rewards. This formalism is used both in decision-theoretic planning (DTP), RL, and other learning problems in stochastic domains [2].

We assume that the process evolves through a sequence of discrete time steps, $t = 1, 2, 3, \ldots$, even if it is possible to extend to the continuous case. These steps

do not need to reflect fixed intervals of real time, but they can refer to arbitrary successive stages of decision-making and acting. At each time step t, the agent detects the *state* of the environment, $S_t \in \mathcal{S}$, based on which it selects an action $A_t \in \mathcal{A}(s)$ for a certain value s of the state (if the action set is the same in all states, we will write it simply as $\mathcal{A}$). Then, as a consequence, the agent receives, one time step later, a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and ends up in a new state S_{t+1}. Thus, given this sequential process, the following sequence, or *trajectory*, emerges:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, \dots \tag{2.1}$$

In a *finite* MDP we have that the set of states $\mathcal{S}$, actions $\mathcal{A}$, and rewards $\mathcal{R}$ have finite cardinalities, respectively $|\mathcal{S}|, |\mathcal{A}|$, and $|\mathcal{R}|$. From now on, we will consider only this kind of MDPs.

Since it is a *finite* MDP, the random variables R_t and S_t have well-defined discrete probability distributions, and since we are considering a *Markov* process, the *Markov property*, also called *memorylessness property*, holds. This means that the *dynamics* of the process across the space of the possible states, also called the *model* of the environment, depends only on the preceding state and action:

$$p(s', r \mid s, a) := \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}\,, \tag{2.2}$$

for specific values $s' \in \mathcal{S}$ and $r \in \mathcal{R}$ occurring at time t, given $s \in \mathcal{S}, a \in \mathcal{A}(s)$.

One could also compute the *transition probabilities*, defined (with a little abuse of notation, since we use the same letter p) as

$$p(s'|s, a) := \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a)\,. \tag{2.3}$$

The *expected rewards* $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to R$ for state–action–next-action triples can be computed as

$$r(s, a, s') := \mathbb{E}\left[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'\right] = \sum_{r \in \mathcal{R}} r \cdot p(s', r|s, a)\,. \tag{2.4}$$

The MDP can also be seen as the tuple $(\mathcal{S}, \mathcal{A}, p, r)$, where $r = r(s, a, s')$ is the expected immediate reward that the agent receives for taking action a in state s [3, 4].

In order to describe how likely it is for an agent to take any given action from any given state, we use the notion of policy. A *policy* is a class of probability distributions $\pi(a|s)$ over an action $a \in A(s)$ for each state $s \in \mathcal{S}$:

$$\pi(a|s) = \Pr\{A_t = a|S_t = s\}\,, \tag{2.5}$$

which describes the probability that, being in that state at that time step t, the agent chooses that action.

The agent's goal is to maximize the *expected return*, o *cumulative reward*, it receives in the long run given a certain policy π:

$$\mathbb{E}[G] = \mathbb{E}_{p(\cdot,\cdot|s_t,a_t),\pi}\left[\sum_{t=0}^{\infty}\gamma^t R_{t+1}\right]\,, \tag{2.6}$$

where $\gamma \in [0,1]$ is the *discount factor*, and it determines how much we value future rewards. For $\gamma = 0$ the agent considers only immediate rewards and discards future ones, while for $\gamma = 1$ the agent will take into account, in the same way, every reward it receives, considering a very long series of events.

Actually, we can take this average over the expected rewards, so as to remove the dependency on the distribution of rewards. This is a distinctive property of the fact that we are working with a *known model of the environment*. So, since we are only interested in optimizing averages, we do not care any longer about what is the actual distribution of the rewards. Thus, we can rephrase the agent goal as

$$\mathbb{E}[G] = \mathbb{E}_{p(\cdot|s_t,a_t),\pi}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t,s_{t+1})\right]\,. \tag{2.7}$$

To reach its goal, the agent must estimate how good it is to be in a given state if it is following a certain policy. This value is stored in the *(state-)value function*. Formally, the value of state s under policy π is

$$V_\pi(s) := \mathbb{E}\left[G_t \mid S_t = s\right] = \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^k R_t \,\middle|\, S_t = s\right],\ \text{for all } s \in \mathcal{S}\,, \tag{2.8}$$

which is the expected return when starting in state s and following policy π thereafter. If there exists a terminal state, which causes the process to terminate when reached, its value is always zero.

Similarly, the goodness of taking action a in state s under policy π is a value stored in the *(state-)action value function*, or *quality*, which is expressed with

$$\begin{aligned} Q_\pi(s,a) &:= \mathbb{E}\left[G_t \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^k R_t \,\middle|\, S_t = s, A_t = a\right], \end{aligned} \tag{2.9}$$

which is the expected return starting from state s, taking action a and following policy π thereafter.

We have that the following relation holds:

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a)\,. \tag{2.10}$$

The value function possesses a fundamental property, which is extensively used in RL: it satisfies the following recursive relation

$$\begin{aligned} V_\pi(s) &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V_\pi(s')\right] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s' \mid s, a)\left[r(s,a,s') + \gamma V_\pi(s')\right], \end{aligned} \tag{2.11}$$

which is called *Bellman equation for* V_π.

Similarly, we have the *Bellman equation for* Q_π:

$$\begin{aligned} Q_\pi(s,a) &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s', a')\right] \\ &= \sum_{s'} p(s' \mid s, a)\left[r(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s', a')\right]. \end{aligned} \tag{2.12}$$

2.3 Partially Observable MDPs

The presentation of this topic will mainly follow the one in [5].

What happens if the agent does not know anymore with full certainty the state of the environment, maybe due to imperfections or limitations in the agent's sensors, or to errors in the interpretation of the environment, or simply to unknown aspects of the environment itself?

If some features of the state are hidden from the agent, and the latter can sense only a part of the real state of the environment, the resultant state signal will no longer be Markovian, breaking a key assumption of most RL methods, like the MDP formulation. Luckily, we can consider an extension of the (fully observable) MDP framework, that can deal with the uncertainty originating from the imperfect states perceived by the agent, or with the uncertain effects of taking an action.

A *partially observable Markov decision process*, or POMDP, is a framework that considers environments that are only partially observable to the agent, but allows anyway for optimal decision-making [3], contrary to the requirement of full knowledge of the environment of MDPs.

The fact that the environment is partially observable can derive mainly from two reasons:

- different states generate the same observation, due to the same sensor reading, caused by the agent's limited knowledge of the environment;
- sensor readings are noisy, so the same state can generate different observations due to different sensor readings.

Thus, we have that the state of the environment is not uniquely identified by the agent's observations, and situations that appear similar to the agent may instead require different actions.

Interestingly, we can consider fully observable MDPs as a special case of POMDPs, in which the observation function deterministically maps each state to its correct unique observation [6].

Since a POMDP is an extension of an MDP, they share many elements. Even in POMDPs the time is divided into different steps, and in each one of them the agent has to take an action. As for MDPs, we will consider discrete and finite models, which are simpler than the continuous ones. So the environment is represented with a finite set of states $\mathcal{S} = \{s_0, s_1, s_2, \ldots, s_N\}$, and a finite set of possible actions is $\mathcal{A} = \{a_0, a_1, \ldots, a_K\}$. But instead of directly perceiving the state in which the environment is, the agent senses an *observation* of it — a signal that depends on the true state of the system but provides only partial information about it. The

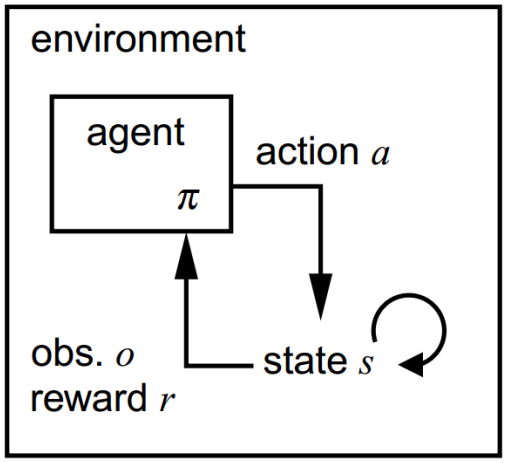


Figure 2.2: The agent-environment interaction in a POMDP [5].

set of observations is discrete and finite: $\mathcal{O} = \{o_0, o_1, \ldots, o_M\}$, and represents all the possible sensor readings the agent can receive. Figure 2.2 represents the interaction between these elements in the POMDP.

If the system is in state s and the agent performs action a, we have that the system transitions to state s' according to the transition probability $p(s'|s,a)$ as in an MDP, but the agent receives observation o' with probability

$$z(a, s', o') = \Pr(o'|a, s')\,, \tag{2.13}$$

using the notation of [6].

Thus, a POMDP can be described as the tuple $(\mathcal{S}, \mathcal{A}, p, r, \mathcal{O}, z)$, where $r = r(s, a, s')$ is the expected immediate reward, as in an MDP [3].

The goal of the agent is, once again, to find the optimal policy π that maximizes the expected return in (2.7). To solve both MDPa and POMDPs, one can use several different methods, which are classified according to the dimensions of the state and action spaces, as well as based on whether the model of the environment is available or not. In fact, there are *tabular methods* — which are exact and use arrays to store value functions, or *approximate solution methods* — which employ function approximators, using limited computational resources; then there are *model-based methods* — which make use of the known model of the environment and of planning in order to find the exact optimal policy, or *model-free methods*

— which are trial-and-error learners.

Often, finding an exact optimal policy for POMDPs can be computationally very expensive, so it is necessary to use approximate methods even for relatively small problems [7]. They are divided into two main classes: *value-function methods* — or *action-value methods* as in [1], that attempt to learn an approximate value function of the so-called belief states (which are probability distributions over the states); and *policy-based methods*, that search for a good policy within a fixed class of parameterized policies. We will focus on the second ones, since it is what we will implement to solve our problem.

2.4 Policy gradient methods

A *policy gradient method* is a policy-based method that attempts to optimize the parameters of a parametrized policy using either *gradient ascent* or *gradient descent* in the parameters' space, based on the expression of the agent's goal. In fact, it tries to maximize (or minimize) the expected return of a policy inside the policy class dictated by the parametrization. This method, then, directly uses the policy to choose the actions, instead of consulting a value function, which however might still be used to learn the parameters.

Policy gradient methods present some perks that make them highly valuable: the parametrized policy allows for direct incorporation of potential environment insights, they can naturally switch from the discrete setting to the continuous one, and they converge to at least a locally optimal policy, even if, as a drawback, the convergence can be very slow and finding the global optimum instead of a local one can be hard [8]. Another distinctive trait of these methods is that they can also naturally handle partial state information. In fact, if a state variable cannot be observed, then we can choose the policy parametrization such that the parameters do not depend on that state variable. Thus, we can directly apply these methods to POMDPs, without having to make any changes.

We will denote the policy's parameters' vector with $\boldsymbol{\theta} \in \mathbb{R}^d$, and we will write,

as in (2.5),

$$\pi(a|s,\boldsymbol{\theta}) = \Pr\{A_t = a|S_t = s;\boldsymbol{\theta}_t = \boldsymbol{\theta}\}\,, \tag{2.14}$$

for the probability of taking action a being in state s at time t with parameters $\boldsymbol{\theta}$ [1]. The policy can be parametrized in any way, as long as it is differentiable with respect to its parameters, which means that the column vector of partial derivatives of π with respect to the components of the parameters' vector $\boldsymbol{\theta}$, $\nabla_{\boldsymbol{\theta}}\pi(a|s,\boldsymbol{\theta})$, exists and is finite for all $s \in \mathcal{S}, a \in \mathcal{A}$ and $\boldsymbol{\theta} \in \mathbb{R}^d$. In particular, as described in [8], for discrete problems the (exponential) soft-max distribution (i.e., Gibbs or Boltzmann distribution) is often used

$$\pi(a|s,\theta) = \frac{e^{\phi(s,a)^\top\theta}}{\sum_{b\in\mathcal{A}} e^{\phi(s,b)^\top\theta}}\,, \tag{2.15}$$

while for continuous problems the Gaussian distribution is used

$$\pi(a|s,\theta) = \mathcal{N}(\phi(s,a)^\top\theta_1, \theta_2)\,, \tag{2.16}$$

where θ_2 is an exploration parameter, and $\phi(s,a) \in \mathbb{R}^d$ is a feature vector characterizing state s and action a [9].

To learn the policy parameters $\boldsymbol{\theta}$, we will use the gradient of some scalar *performance measure* $J_\pi(\boldsymbol{\theta})$ — which can also directly be the expected return — with respect to the policy parameters. If the performance $J_\pi(\boldsymbol{\theta})$ measures the rewards that the agent receives, we want to maximize it, so we update the parameters approximating gradient *ascent* in $J_\pi(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta}_k)\,. \tag{2.17}$$

Instead, if the performance measures the costs that the agent encounters, we seek to minimize it, so we use approximate gradient *descent* in $J_\pi(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta}_k)\,. \tag{2.18}$$

In both cases, $\alpha > 0$ is a *step-size parameter* or the *learning rate*.

As done before, we will treat the episodic case, for which we define the *performance measure* $J_\pi(\boldsymbol{\theta})$ as the average value of the initial states of the process:

$$\begin{aligned} J_\pi(\boldsymbol{\theta}) &= \sum_s \rho_0(s) V_{\pi_{\boldsymbol{\theta}}}(s) \\ &= \sum_s \rho_0(s) \sum_a \pi(a|s) Q_{\pi_{\boldsymbol{\theta}}}(s,a)\,, \end{aligned} \tag{2.19}$$

where we define as $\rho_0(s')$ the probability that an episode begins in state $s' \in \mathcal{S}$.

Let us define $\eta_\pi(s')$ as the average number of time steps that the agent spends in state s' before the process terminates. Time is spent in a state s' if episodes start in that state s', or if the environment transitions into s' from a previous state s in which time is spent. Thus, we have that the formula for η_π is

$$\eta_\pi(s') = \rho_0(s') + \sum_s \eta_\pi(s) \sum_a \pi(a|s) p(s'|s,a), \text{ for all } s' \in \mathcal{S}. \tag{2.20}$$

This is a *linear system* of equations, one for each state $s' \in \mathcal{S}$, and it can be solved for the expected number of visits $\eta_\pi(s)$.

The *policy gradient theorem* provides an analytic expression for the gradient of the performance J with respect to the policy parameters, which is what we need to approximate for gradient ascent in (2.17) (or descent in (2.18)):

$$\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta}) = \sum_s \eta_\pi(s) \sum_a Q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s). \tag{2.21}$$

See [1] for the proof in the episodic case, which involves just elementary calculus and re-arranging of terms in the expression of the value function V_π.

Thus, solving the expression for (2.21) and using it in equations (2.17) – (2.18) allows optimizing the parameters of the policy, building an optimal policy in the chosen parametrization class. Note that, in order to find both η_π and Q_π, we need to solve two linear systems, (2.12) and (2.20), and, moreover, they depend on the policy π, thus they must be solved at every iteration.

2.4.1 Natural policy gradient

Ref. [10] introduces the *natural gradient learning method*, which is a different way of performing a gradient descent for supervised learning problems, using a different

formula for the gradient. This method is proven to be statistically efficient and does not get stuck in plateaus, as the conventional stochastic gradient learning does. This is because the ordinary gradient of a function, in some parameters spaces, does not identify its steepest direction, while the natural gradient does.

This method has also been applied in RL in policy gradient algorithms [11], by replacing the gradient $\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$ with the *natural gradient* $\widetilde{\nabla}_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$ when performing the parameters' updates. The difference is that the first follows the steepest direction in the parameter space, while the latter follows the steepest direction with respect to the Fisher metric, given by

$$\widetilde{\nabla}_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta}) = F^{-1}(\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta}) \tag{2.22}$$

where $F(\boldsymbol{\theta})$ denotes the *Fisher information matrix*, which we are going to define in the following. Since it can be proven that the angle between the two different gradients is never larger than 90°, the convergence to the next local optimum is guaranteed. The natural policy gradient turns out to be significantly more efficient than normal gradients, since the convergence to the local minima is much faster.

The *Fisher information* is a metric which measures how much information a random variable X carries about an unknown parameter θ that parametrizes the probability density function $f(X;\theta)$ of the variable X itself. It has formula

$$\mathcal{F}(\theta) = \mathbb{E}_\theta \left[\left(\frac{\partial}{\partial \theta} \log f(X;\theta) \right)^2 \right] = \int_{\mathbb{R}} \left(\frac{\partial}{\partial \theta} \log f(x;\theta) \right)^2 f(x;\theta)\, dx\,. \tag{2.23}$$

If the density function $f(x;\theta)$ is at least twice differentiable with respect to the parameter θ, and under certain regularity conditions, then the Fisher information may also be written as

$$\mathcal{F}(\theta) = -\mathbb{E}_\theta \left[\frac{\partial^2}{\partial \theta^2} \log f(X;\theta) \right]. \tag{2.24}$$

The identity subsists as

$$\begin{aligned} \frac{\partial^2}{\partial \theta^2} \log f(X;\theta) &= \frac{\partial}{\partial \theta} \left(\frac{1}{f(X;\theta)} \frac{\partial}{\partial \theta} f(X;\theta) \right) \\ &= -\frac{1}{f(X;\theta)^2} \left(\frac{\partial}{\partial \theta} f(X;\theta) \right)^2 + \frac{1}{f(X;\theta)} \frac{\partial^2}{\partial \theta^2} f(X;\theta)\,, \end{aligned}$$

but

$$\mathbb{E}_\theta\left[\frac{1}{f(X;\theta)}\frac{\partial^2}{\partial\theta^2}f(X;\theta)\right] = \int_{\mathbb{R}}\frac{1}{f(X;\theta)}\frac{\partial^2}{\partial\theta^2}f(X;\theta)\cdot f(X;\theta)dx = \int_{\mathbb{R}}\frac{\partial^2}{\partial\theta^2}f(X;\theta)dx$$
$$= \frac{\partial^2}{\partial\theta^2}\int_{\mathbb{R}}f(X;\theta)dx = \frac{\partial^2}{\partial\theta^2}1 = 0\,.$$

Thus, we have that $\mathcal{F}(\theta)$ provides a measure of the intensity of the local curvature of the function $\log f(X;\theta)$ in a neighborhood of θ: the larger $\mathcal{F}(\theta)$, i.e., the more pronounced the curvature of $\log f(X;\theta)$, the more concentrated the function $\log f(X;\theta)$ is around θ, and the more information we have.

Consider the specific case of the soft-max policy (2.15), where instead of defining the feature vector $\phi(s,a)$, we take a parameter for each state s and action a, $\theta_{s,a}$, obtaining a parameters' matrix $\boldsymbol{\theta} = (\theta_{s,a})_{s\in\mathcal{S},a\in\mathcal{A}}$ (actually, it is a vector, but the visualization as a matrix is more convenient):

$$\pi(a|s;\boldsymbol{\theta}) = \frac{e^{\theta_{s,a}}}{\sum_{d\in\mathcal{A}}e^{\theta_{s,d}}}\,. \tag{2.25}$$

Following [12] (appendix A.4), we then define the Fisher information matrix F of the previous policy $\pi_{\boldsymbol{\theta}}$ as $F(\boldsymbol{\theta}) = (F_{s,a,s',b})_{s,s'\in\mathcal{S},a,b\in\mathcal{A}}$, where, for a specific state s,

$$F_{a,b} = \sum_{c\in\mathcal{A}}\left(\frac{\partial}{\partial_{\theta_{s,a}}}\log\pi(c|s;\boldsymbol{\theta})\right)\left(\frac{\partial}{\partial_{\theta_{s,b}}}\log\pi(c|s;\boldsymbol{\theta})\right)\pi(c|s;\boldsymbol{\theta})\,, \tag{2.26}$$

in which we used the second equivalence of (2.23).

First of all, we have that the gradient of the policy, $\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}$, has dimensions $|\mathcal{S}|\times|\mathcal{A}|\times|\mathcal{S}|\times|\mathcal{A}|$, and in particular the derivative of the soft-max policy, for a fixed state s, is

$$\begin{aligned}\frac{\partial}{\partial\theta_{s,a}}\pi(c|s;\boldsymbol{\theta}) &= \frac{\partial}{\partial\theta_{s,a}}\left(\frac{e^{\theta_{s,c}}}{\sum_{d\in\mathcal{A}}e^{\theta_{s,d}}}\right)\\ &= \frac{\delta_{a,c}\cdot e^{\theta_{s,c}}\cdot\sum_{d\in\mathcal{A}}e^{\theta_{s,d}} - e^{\theta_{s,c}}\cdot e^{\theta_{s,a}}}{(\sum_{d\in\mathcal{A}}e^{\theta_{s,d}})^2}\\ &= \left(\frac{\delta_{a,c}\cdot\sum_{d\in\mathcal{A}}e^{\theta_{s,d}}}{\sum_{d\in\mathcal{A}}e^{\theta_{s,d}}} - \frac{e^{\theta_{s,a}}}{\sum_{d\in\mathcal{A}}e^{\theta_{s,d}}}\right)\frac{e^{\theta_{s,c}}}{\sum_{d\in\mathcal{A}}e^{\theta_{s,d}}}\\ &= (\delta_{a,c} - \pi(a|s;\boldsymbol{\theta}))\,\pi(c|s;\boldsymbol{\theta})\,.\end{aligned}$$

Thus, in this case, we have that

$$\begin{aligned}
F_{a,b} &= \sum_{c\in\mathcal{A}} \left(\frac{\partial}{\partial_{\theta_{s,a}}} \log \pi(c|s;\boldsymbol{\theta})\right)\left(\frac{\partial}{\partial_{\theta_{s,b}}} \log \pi(c|s;\boldsymbol{\theta})\right)\pi(c|s;\boldsymbol{\theta}) \\
&= \sum_{c\in\mathcal{A}} \left(\frac{1}{\pi(c|s;\boldsymbol{\theta})}\frac{\partial}{\partial_{\theta_{s,a}}}\pi(c|s;\boldsymbol{\theta})\right)\left(\frac{1}{\pi(c|s;\boldsymbol{\theta})}\frac{\partial}{\partial_{\theta_{s,b}}}\pi(c|s;\boldsymbol{\theta})\right)\pi(c|s;\boldsymbol{\theta}) \\
&= \sum_{c\in\mathcal{A}} \left(\frac{1}{\pi(c|s;\boldsymbol{\theta})}\big(\delta_{a,c} - \pi(a|s;\boldsymbol{\theta})\big)\,\pi(c|s;\boldsymbol{\theta})\right)\left(\frac{1}{\pi(c|s;\boldsymbol{\theta})}\big(\delta_{b,c}\right. \\
&\qquad \left. - \pi(b|s;\boldsymbol{\theta})\big)\,\pi(c|s;\boldsymbol{\theta})\right)\pi(c|s;\boldsymbol{\theta}) \\
&= \sum_{c\in\mathcal{A}} \big(\delta_{a,c} - \pi(a|s;\boldsymbol{\theta})\big)\big(\delta_{b,c} - \pi(b|s;\boldsymbol{\theta})\big)\pi(c|s;\boldsymbol{\theta}) \\
&= \sum_{c\in\mathcal{A}} \Big[\big(\delta_{a,c} - \pi(a|s;\boldsymbol{\theta})\big)\delta_{b,c}\pi(c|s;\boldsymbol{\theta}) - \big(\delta_{a,c} - \pi(a|s;\boldsymbol{\theta})\big)\pi(b|s;\boldsymbol{\theta})\pi(c|s;\boldsymbol{\theta})\Big] \\
&= \big(\delta_{a,b} - \pi(a|s;\boldsymbol{\theta})\big)\pi(b|s;\boldsymbol{\theta}) - \pi(b|s;\boldsymbol{\theta})\pi(a|s;\boldsymbol{\theta}) + \pi(a|s;\boldsymbol{\theta})\pi(b|s;\boldsymbol{\theta})\sum_{c\in\mathcal{A}}\pi(c|s;\boldsymbol{\theta}) \\
&= \big(\delta_{a,b} - \pi(a|s;\boldsymbol{\theta})\big)\pi(b|s;\boldsymbol{\theta})\,.
\end{aligned}$$

This gives us, for a specific state s,

$$F_{a,b} = \big(\delta_{a,b} - \pi(a|s;\boldsymbol{\theta})\big)\pi(b|s;\boldsymbol{\theta}) = \frac{\partial}{\partial\theta_{s,a}}\pi(b|s;\boldsymbol{\theta})\,, \tag{2.27}$$

which means that

$$F = \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}\,, \tag{2.28}$$

and therefore we have that

$$\widetilde{\nabla}_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}} = F^{-1}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}} = I_{(\mathcal{S},\mathcal{A})\times(\mathcal{S},\mathcal{A})}\,. \tag{2.29}$$

This is a very important result since it allows avoiding computing, and then inverting, the information matrix F, saving quite some computational time.

Chapter 3

Data and bisection algorithms

3.1 Data structures

Our first objective is to model Trieste's power grid, in order to have a computational representation on which we can develop the algorithms that mimic the restoration process and find the fault on the grid.

The dataset we use consists of some shapefiles which contain the power grid elements that are present in the GIS platform of the AAA company. The shapefile format is a geospatial vector file format, which can also store the geometric location of an object for geographic information system (GIS) software. It stores the data as primitive geometric shapes, like points, lines, and polygons. Thanks to these shapes and the data attributes that are linked to each one of them, above all the position, it is possible to create the representation of the geographic data.

The company GIS also contains, among others, all the elements of the power grid, and every night it performs a snapshot of the current state, saving a different shapefile for every type of object. We use the shapefiles of every component in medium voltage: electrical cables, breakers, overhead breakers, busbars, cable joints, connections, conjunctors, pods, and also of primary substations, secondary substations, and transformers.

We proceed by refining, processing, and analyzing the data. First of all, we

transform the shapefiles in Parquet files (a columnar data storage format of the Apache Hadoop ecosystem), compressing them with snappy (a fast data compression library, developed by Google and written in C++). In fact, loading and analyzing the shapefiles each time is very time-consuming, while with snappy compressed Parquet files we can really speed up the process.

After having cleaned a bit the dataset, we remove all the elements that are flagged with something different from "in service" or "out of service" (but functioning), like "not functioning", or "in construction", or others, in order to have only the elements that are effectively used. Note that the extra cables used to create the meshed power grid are flagged as "in service". Since the database does not contain any information on whether a substation is remotely controlled, we have to look for it in the electrical diagrams, and then update the dataset accordingly. Eventually, all the remotely controlled substation and each of its breakers will be effectively flagged as remotely controlled.

We use two other pre-elaborated databases, one of which connects the pods in low voltage to their transformers, and the other computes the power in kW of each pod (always in low voltage), in order to be able to compute the number of users under each transformer and the overall power of the pods.

Then we create the different data structures we need to properly model the power grid. When deciding which one would be the most appropriate, it comes natural to use graphs, since we have different elements that are connected to each other through cables. So we construct three different graphs, each refining the previous one.

3.1.1 The circuit graph

In the first graph, we put together every component of the shapefiles, creating the *circuit graph*, which is an unrefined graph in which every electrical component of the power grid appears. It is an undirected graph since we only use it to know the connections among the various elements.

We design a class that creates the circuit graph of a power grid from the dataset, adding the nodes and the edges to the graph in the right way using

the electric elements we selected. We have that the nodes are the primary and secondary substations, transformers, busbars, breakers, overhead breakers, cable joints, and pods, while the edges are electrical cables, conjunctors, and connections. We are able to connect the elements among them thanks to the dataset field `idsap`, a unique identifier that is associated with each element, and thanks to the fields that specify the connections among the components. We put in each node the useful information we have about that element, like its class (the type of object), `idsap`, code (another element identifier, but not unique — we can see them for the substations in Figure 1.2), position (longitude and latitude in the GPS coordinates), and other class-specific information, like the voltage and the number of underlying users for the transformers, or whether they are enabled for the breakers (whether they are open in the standard setup). We use a specific node for each substation, even if the substation is merely the set of its components, because we need to save the corresponding information in the graph, and this was a simple solution. Some manual modifications are needed in order to fix the graph appropriately.

After that, we create a class that simplifies a circuit graph, removing the cable joints. *Cable joints* are simply used to connect two branches of a cable but, for us, they have no specific role. So, in our model, they only overburden the graph, adding more nodes that are not relevant. In fact, of around 10000 nodes, 3000 are cable joints. Therefore, if a cable joint simply connects two cables, we remove it, and we create a unique cable, aggregating the information of the two. We keep only those cable joints that connect three or more cables. In this way, we also remove cable joints that have only one cable attached to them, instead of two, which are dead ends.

How do we do it? We start from the complete graph, and then we prune it. We iteratively remove all the cable joints, in order to be sure of removing them all. We first create a list of nodes to be removed, which is filled with only the cable joints with node degree 2 or less. Then we scan this list, and if a cable joint connects two cables, we aggregate the information of these edges in a new one, and we remove the joint (note that, when we remove a node, all the edges that are

connected to it will be removed as well). Instead, if the node degree is less than 2, we directly remove the joint since it is a dead end. In this way, we are able to simplify our initial graph.

3.1.2 The electrical graph

Having only the electrical components of the power grid is not sufficient, though; we also need to know the direction of the power flow. To model that, we use the *electrical graph*, which is a graph that encodes the electricity flow that passes through the elements of the circuit graph in the standard setup, when all the faulty elements are restored. Thus, it is a directed graph, since now we need to know also the specific direction of the power flow.

We construct it by first adding all the nodes of the simplified circuit graph, and then adding the edges — which are electrical cables, conjunctors, and connections — following the direction of the electricity flow. We start from every busbar of the primary substations, since they are the starting points of each electrical line, and we follow each connection among the elements, adding it to the graph in the right direction (pointing from the first element we encountered to the second). Since we are modeling the power flow in the standard setup, if we encounter a cable interrupted by an open breaker — part of the ones used to create the meshed power grid — we do not add it as an edge. In this way, we do not introduce loops in our graph. After each edge is added (or ignored), we have finished creating our electrical graph.

3.1.3 The substations graph

The last graph we create is the *substations graph*, a high-level, directed graph that models only the substations and their connections. We construct it by pruning the electrical graph of the other elements, so as to be able to inherit the connections among the substations and the direction of the power flow. In fact, a substation does not know either which are its elements, either which substations are before or after, only its components are connected among them. What we do is to cycle

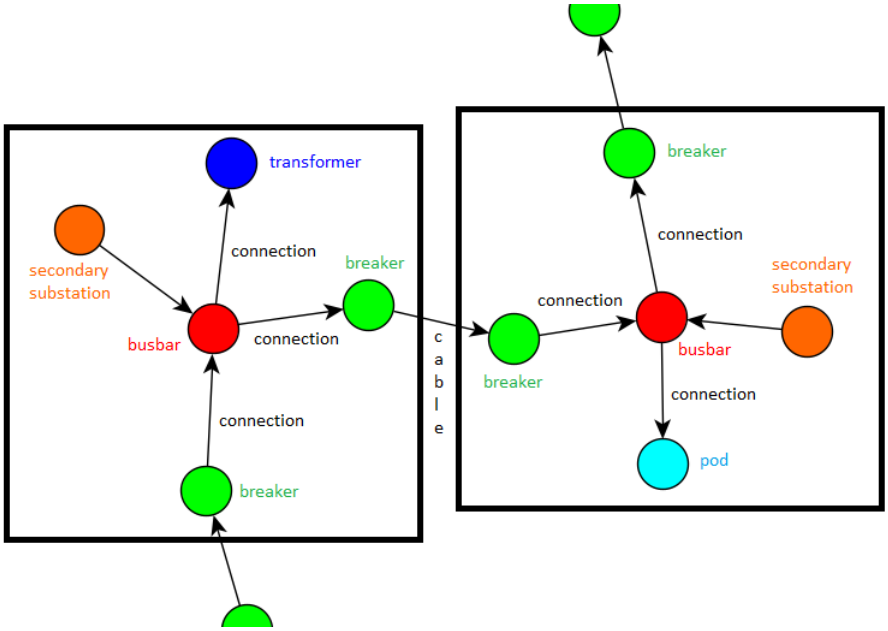


Figure 3.1: Detail of the elements of the electrical graph.

over all the elements, and for each type, we perform a specific action. Pods and transformers can be directly removed, since they are leaves in the electrical graph, so they can be deleted easily. To remove busbars and breakers, we create a new edge between their predecessor and their successor (being careful of creating it in the right direction), named with the `idsap`s of the two nodes at its ends separated by a dash, and then we remove them, which automatically removes also the edges connected to them. If everything is done in the right way, at the end we will have the edges among the substations labeled with their `idsap`s separated by a dash, first the parent's one and then the children's one. Then we compute the number of underlying users of each substation, summing the ones of each transformer or pod (in medium voltage) connected to it, and we store it in the node information.

In Figure 3.2 we can see a part of the substations graph, which is the computational model of the electrical diagram of Figure 1.2. Every node carries a lot of information about the substation, like its code, `idsap`, position, whether it is remotely controlled or not (labeled in bold and with an `R` in the figure if it is), and the number of users underneath it.

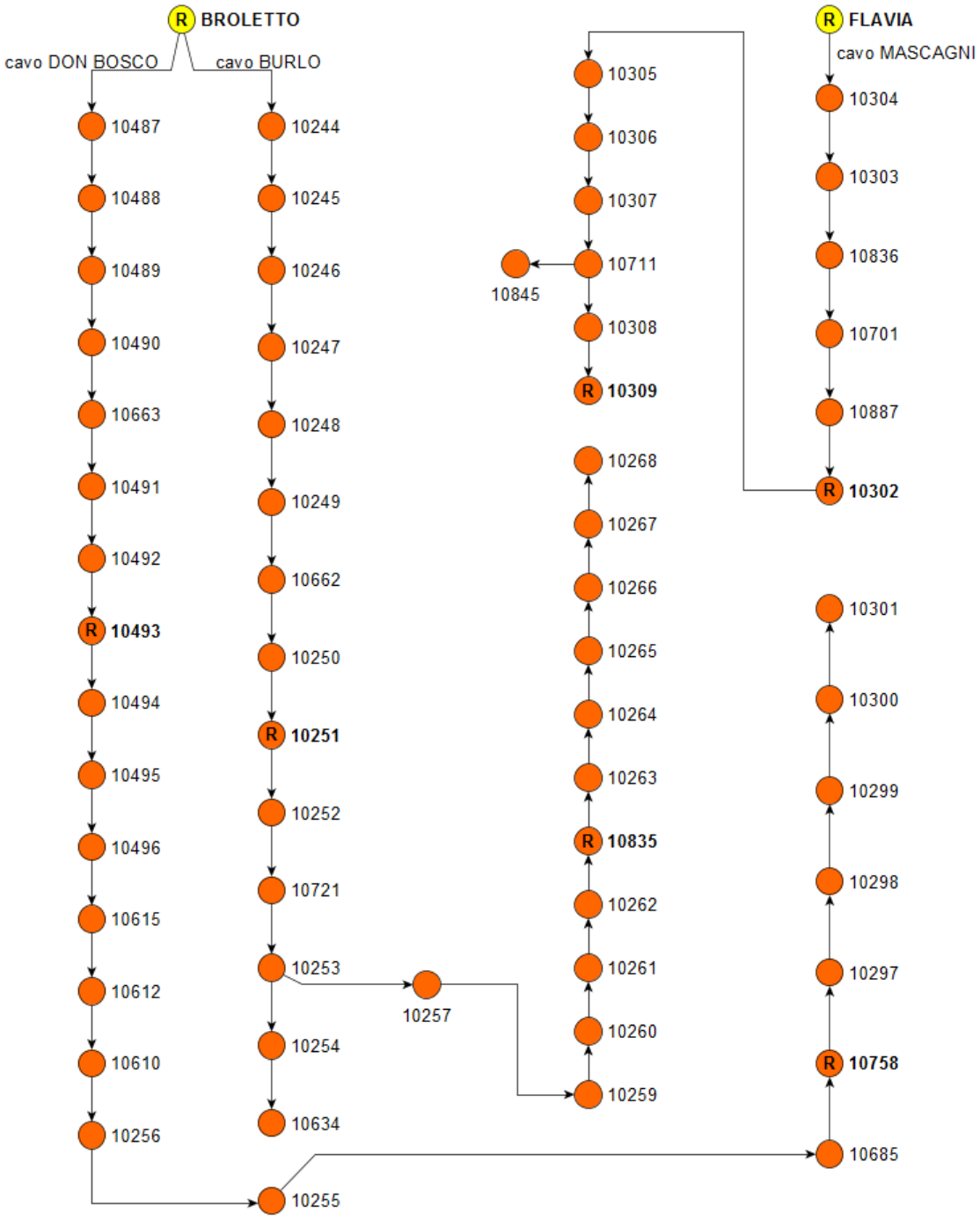


Figure 3.2: Computational model of the electrical diagram of Figure 1.2. The substations with a bold label and with an R are the remotely controlled ones.

3.2 Simulator

The simulator is a class that creates a fault in a specific point of the power grid, or is given the position of a fault, and then using the substations graph can tell the questioner – which will be one our algorithms — both if the fault is before or after the current substation in which the technician is located, and how many substations can be reconnected from that substation. It encodes the information available to the technician through the different methods at their disposal to discover if the fault is before or after the substation in which they are, and the results of their actions in a substation.

To discover if the fault is before or after the currently visited substation, a function of the simulator computes the predecessors and the successors of the current substation; then it scans the graph, starting from each successor, looking for the fault, and, if it finds it, it returns the successor from which it started. If it finishes the graph, arriving in all the leaves, without finding the fault, it scans the graph from each predecessor upwards; and when it finds the fault it returns the predecessor from which it started. In this way, the simulator returns either the subsequent substation of the current one, indicating that the fault is after it, or the previous substation of the current one, indicating that the fault is before it.

Instead, to compute the list of substations that can be reconnected by operating in the current one, the function takes the output of the previous function, computes the list of predecessors of the current substation, and if the fault is in it, it starts from the current substation and visits all the subsequent ones till the end, adding all of them to a set which is returned at the end of the computation. If the fault is not in the predecessors, it is in the successors, so a traversal of the graph is made from the first remotely controlled substation until the current substation is reached, adding all these substations to a set and returning it at the end of the computation.

3.3 Algorithms

At the moment, in all our algorithms we use a simplified version of the power grid, which does not consider electrical line forks, in order to simplify the computations. Thus, we only feed our algorithms with rectilinear power lines.

3.3.1 Bisection

The bisection algorithm, which is actually a binary search algorithm, plus their experience, is what the technicians currently use to solve a fault and reconnect all the disconnected substations left out after the initial operations. We create a class that models this method, although of course not exactly, since we don't have their experience at our disposal. Our bisection algorithm, in fact, simply takes the number of currently disconnected substations and visits the one in the middle, always selecting the one on the left in case of ties. Then it uses the simulator to compute the set of remaining disconnected substations and selects another action until this set is empty. The technicians, instead, do not choose exactly the substation in the middle, but bend this rule based on many different factors, like how easy it is to enter a substation, whether it is day or night — which also affects the first condition, the weather conditions, etcetera.

In order to be able to compare all our algorithms, we compute the performance measure $J_\pi(\boldsymbol{\theta})$ also for the bisection algorithm. So, first of all, we have to compute the policy matrix. For each state, we compute the substation that the bisection would choose in that situation, and we encode it in the corresponding policy row as a one hot-vector in which that action has coefficient 1, while all other actions that can be taken in that state have a 0, as well as the inadmissible actions. Then we compute the matrix of Q_π using equation (2.12) (in section 4.2 we will describe in detail how we do it) and we use it to compute the performance measure using equation (2.19).

3.3.2 Bisection improved

Since the bisection algorithm, in a tie, blindly chooses the substation to the left, we were curious about how an algorithm slightly better at handling these ties would perform. So we created a bisection algorithm that, when a tie happens, chooses to visit the substation nearer to the current position of the technician. The algorithm iteratively chooses an action and uses the simulator to compute the set of remaining disconnected substations until this set is empty, which means that all the substations have been reconnected, and therefore the fault is solved.

Also for this algorithm, we computed a policy matrix, using one-hot vectors as its rows to indicate which substation it would choose in that state. Given this matrix, we were able to compute the performance measure $J_\pi(\boldsymbol{\theta})$ like for the bisection.

3.3.3 Weighted bisection

The weighted bisection is yet another variation of the bisection algorithm, which determines the halfway substation to be visited according to the weights assigned to every substation. The weight of a substation is related to the cost of a fault: it is computed as the distance of the substation from the current position of the technician — as the time it takes, in seconds, to go there — multiplied by the number of underlying users of that substation. Thus, the algorithm computes the weight of each currently disconnected substation and decides the substation to visit by taking the one which minimizes the absolute value of the difference between the sum of the weights of the previous substations and the sum of the weights of the subsequent ones.

An efficient way to do it is to compute two vectors of the cumulated sums of the weights: one by scanning the substations from left to right, and the other one by scanning them from right to left (the latter done by flipping the initial vector, calculating the vector of the cumulated sums from left to right, and then flipping it again). Then we take the absolute value of their difference and we look for the index with the lowest value.

Chapter 4

Problem modelling and implementation

In this chapter, we are going to formulate our problem using the frameworks that we saw in chapter 2: we will define the basic elements, and then we will develop the formulas and the algorithm that we need to solve it.

4.1 The mathematical model

After a fault occurs and after the technician, together with the remote control room, restricts it to a limited number of non-remotely-controlled substations, the problem consists in visiting, and thus reconnecting, these substations in an order that minimizes the cost of the fault. We define the *cost of the fault* as the amount of time each underlying user of each substation remains disconnected.

Thus, we are given a set of initially disconnected substations $\mathcal{C}$, with cardinality $|\mathcal{C}| = N$, between two remotely controlled substations, where these last ones will not be included in the set, since they are already reconnected. Looking at the electrical diagram of Trieste's power grid, we notice that the number of substations between two remotely controlled ones is always less than 20, so we can say that in our problem $N < 20$. Luckily, this restricts the dimension of the problem.

4.1.1 Elements of the POMDP

Given that we don't know the position of the fault, but we have to find it while we reconnect the substations, we cannot use an MDP to model this problem; instead, we will use a POMDP. The *agent* is the technician that has to decide which substation to visit at each step, while the *environment* includes everything else, among which the possible positions of the fault, and the set of disconnected substations, and their positions and distances.

We define the *state* s of the environment as the tuple

$$s = (x_g, v_k, \{v\}) \tag{4.1}$$

where x_g is the position of the fault, $v_k \in \mathcal{C}$ is the substation at which the technician is currently located, and $\{v\}$ is the set of substations still disconnected after the technician operates in the current substation v_k. The set $\{v\}$ can also be chosen to be the set of substations already reconnected, since they are complementary with respect to the set $|\mathcal{C}|$, but we chose to use the disconnected substations to ease the notation. Since the position of the fault is unknown, the variable x_g is *hidden*, while the variables v_k and $\{v\}$ are *observable*. When the fault occurs, the technician can be everywhere: at home if it happens in the middle of the night, at the company, or on the go. So we introduce an extra dummy substation, called substation 0, which is the position of the technician when the fault occurs. Given this, we have that the *initial state* is always of the form $s_0 = (x_g, 0, \mathcal{C})$, thus we have different initial states, one for every possible position of the fault. Instead, a *terminal state* is of the form $s_t = (x_g, v_k, \varnothing)$. If the fault is localized on an electrical cable, then v_k is one of the two substations at the ends of that faulty cable, so we have two different terminal states; while if the fault is in a substation, v_k is that exact substation, so the terminal state is only one. We notice that the initial cost has a random component, which depends on the position of the technician when the fault occurs, so on the position of the substation 0. To remove this randomness, we could also impose that the technician position is always at the company, but we chose not to do that, in order to be closer to what really happens.

We then define the *observation* o that the agent senses from the environment

as

$$o = (v_k, \{v\}) \,, \tag{4.2}$$

and we will also write the state as $s = (x_g, o)$, where $o = (v_k, \{v\})$ is the observation itself. We define the observable o to be a function of s:

$$\begin{array}{cccc} o(s): & \mathcal{S} & \to & \mathcal{O} \\ & s = (x_g,\, o = (v_k, \{v\})\,) & \mapsto & o = (v_k, \{v\}) \end{array} \,. \tag{4.3}$$

For different states $s = (x_g,\, o = (v_k, \{v\})\,)$ that differ only by the position of the fault x_g, this function associates the same observable $o = (v_k, \{v\})$. Thus, we have that o is an equivalence class for s. For brevity, we will often keep this dependency implicit, and write simply o instead of $o(s)$, meaning that $o = o(s)$. In particular, the observation of an initial state $s_0 = (x_g, 0, \mathcal{C})$ is $o_0 = (0, \mathcal{C})$.

We define the *action* a that the agent can do as the choice of the specific substation the technician will visit as the next step, thus we have that $a \in \mathcal{A} = \mathcal{C}$. Actually, since the technician visits only disconnected substations, and never visits already visited substations, we have that $a \in \{v\}$, if we are in state $s = (x_g, v_k, \{v\})$. Thus, we have that the set of available actions depends on the current state:

$$a \in \mathcal{A}\big(s = (x_g, v_k, \{v\})\big) = \{v\} \,. \tag{4.4}$$

We said that this is a problem with terminal states, which occur when we reconnect all the substations. A terminal state will always be reached, since with every action we visit a substation and can reconnect it, so at the very least we remove that substation from the set of disconnected substations. Actually, if we are lucky, each time we can remove half of the substations from the set of disconnected substations. We are therefore positive that the process terminates. So, for this specific problem, it doesn't make sense to introduce a discount factor γ (therefore, we have that $\gamma = 1$ in the formulas of chapter 2).

Given all that, we have that the *next state* s' of the environment is

$$s' = (x_g, v_{k+1} = a, \{v'\}), \tag{4.5}$$

where $\{v'\}$ is the set of disconnected substations after the technician operates in the substation v_{k+1}. Since the technician can always at least reconnect the substation

they visit, the set of disconnected substations decreases after each action, so we have that $\{v'\} \subseteq \{v\} \backslash a$.

Finally, we define the *expected reward* as the *cost* of going to a certain substation (as the time, in seconds, it takes to go there from where the technician is) multiplied by the number of disconnected users. Let's define as $d_{v_k,v_{k+1}}$ the time in seconds to go from the substation v_k to the next substation v_{k+1}, and as n_k the number of users still disconnected *before* operating in the substation v_{k+1}. So if we are in a state $s = (x_g, v_k, \{v\})$, we make an action a, and we end up in a state $s' = (x_g, v_{k+1} = a, \{v'\})$, we have that the number of disconnected users is

$$n_k = \sum_{v \in \{v\}} u_v, \tag{4.6}$$

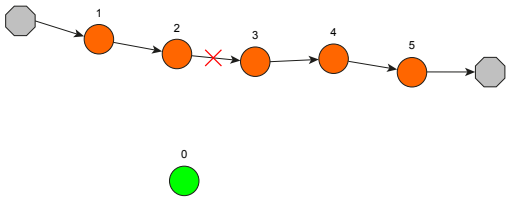
where u_v is the number of users underneath the substation v. So the expected reward has the following formula:

$$r(s, a, s') = d_{v_k,a} \cdot n_k = d_{v_k,a} \cdot \sum_{v \in \{v\}} u_v \,, \tag{4.7}$$

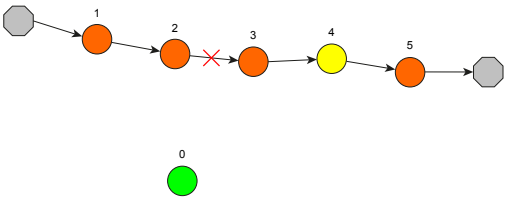
even if it actually depends only on the previous state s and the action taken a, but we will keep the term s' to be consistent with the formulas of [1]. For now, in the cost we will ignore the cost of establishing if the fault is before or after the substation in which the technician is, which is complicated and might raise the total cost significantly. This is because we cannot properly model these costs, due to a lack of data on them. To improve the computation of the cost, we need to carefully take note of the operations the technicians perform when a fault occurs, and then expand the model. To carry out the data collection, one possibility is to implement a serverless Telegram bot using AWS.

In Figure 4.1 we can see (part of) a trajectory (we miss the rewards) of our POMDP, using a set of fictional substations.

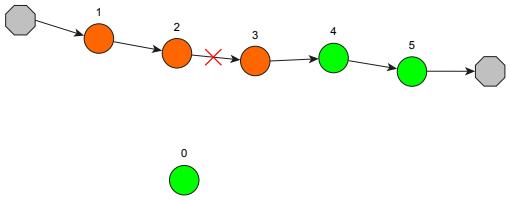
The environment is *deterministic*, so given an admissible action a, we will surely perform it and end up in the state to which that action leads. We can say that there are no execution errors, and we will always do what we want to do. This means that the next state s' is a function of the previous state s and the



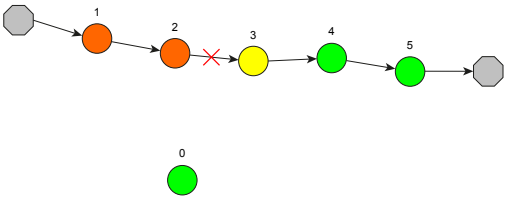
(a) A fault has occurred in 2-3. We are in substation 0 and all the substations are disconnected (orange). Initial state: $s_0 = (\text{2-3}, 0, \mathcal{C} = \{1, 2, 3, 4, 5\})$.



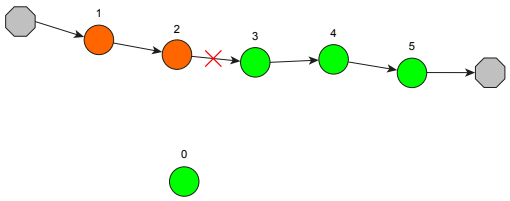
(b) We visit substation 4 (yellow). Action: $a_0 = 4$.



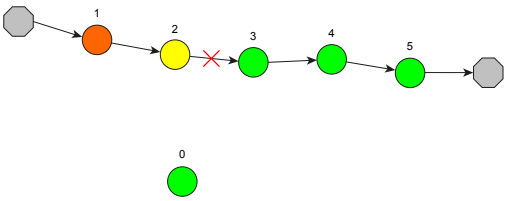
(c) We reconnect substations 4 and 5 (green). State: $s_1 = (\text{2-3}, 4, \{1, 2, 3\})$.



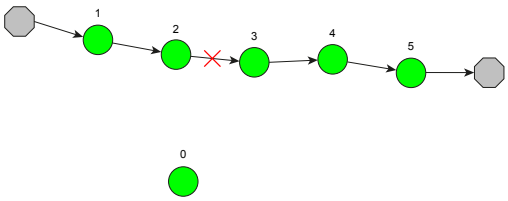
(d) We visit substation 3 (yellow). Action: $a_1 = 3$.



(e) We reconnect substation 3 (green). State: $s_2 = (\text{2-3}, 3, \{1, 2\})$.



(f) We visit substation 2 (yellow). Action: $a_2 = 2$.



(g) We reconnected substations 1 and 2 (green). All the substations are reconnected. Terminal state: $s_3 = (\text{2-3}, 2, \varnothing)$.

Figure 4.1: Fictional example of the sequence of actions of the technician, from the occurrence of the fault to its resolution. The fault is in the electrical cable between substations 2 and 3, and the gray octagonal substations are the remotely controlled ones, while the round substations are the disconnected ones, except substation 0, which is the initial position of the technician.

action a performed: $s' = \sigma(s, a)$. So, mathematically, we have that the *transition probability* is

$$p(s' \mid s, a) = \mathbb{I}\big(s' = \sigma(s, a)\big) = \delta_{s', \sigma(s,a)} = \begin{cases} 1 & \text{if } s' = \sigma(s, a) \\ 0 & \text{otherwise} \end{cases}, \tag{4.8}$$

where $\mathbb{I}$ is the characteristic function of a set, and δ is the Kronecker delta. In our specific case, the transition probability is equal to 1 only when, starting from state $s = (x_g, v_k, \{v\})$, the new substation v_{k+1} of the next state $s' = (x_g, v_{k+1}, \{v'\})$ is equal to the action $a \in \{v\}$ that we took, so:

$$p(s'|s, a) = \begin{cases} 1 & \text{if } v_{k+1} = a \\ 0 & \text{if } v_{k+1} \neq a \end{cases}. \tag{4.9}$$

4.1.2 Policy parametrization

Given that we are in a situation of partial observability, as we don't know the position of the fault, the *policy* depends only on the observables and not on the state, so it doesn't know where the fault is. Let's define a parameterized policy using a simple *tabular* parameterization, in which we have a parameter $\theta_{o,a}$ for each observable o and action a pair:

$$\boldsymbol{\theta} = (\theta_{o,a})_{o \in \mathcal{O}, a \in \mathcal{A}} = \begin{pmatrix} \theta_{o_1,a_1} & \theta_{o_1,a_2} & \cdots & \theta_{o_1,a_N} \\ \theta_{o_2,a_1} & \theta_{o_2,a_2} & \cdots & \theta_{o_2,a_N} \\ \vdots & & & \vdots \\ \theta_{o_{|\mathcal{O}|},a_1} & \theta_{o_{|\mathcal{O}|},a_2} & \cdots & \theta_{o_{|\mathcal{O}|},a_N} \end{pmatrix} \tag{4.10}$$

(where we recall that N is the number of substations initially disconnected, so the number of possible actions: $|\mathcal{A}| = |\mathcal{C}| = N$).

Then we use the soft-max, or Boltzmann, distribution of (2.15) to construct the policy:

$$\pi\Big(a \mid o(s) = (v_k, \{v\}); \boldsymbol{\theta}\Big) = \frac{e^{\theta_{o,a}}}{\sum_{b \in \{v\}} e^{\theta_{o,b}}}. \tag{4.11}$$

So we have that also the policy is a matrix:

$$\pi\Big(a \mid o(s);\boldsymbol{\theta}\Big) = \begin{pmatrix} \pi(a_1|o_1;\boldsymbol{\theta}) & \pi(a_2|o_1;\boldsymbol{\theta}) & \cdots & \pi(a_N|o_1;\boldsymbol{\theta}) \\ \pi(a_1|o_2;\boldsymbol{\theta}) & \pi(a_2|o_2;\boldsymbol{\theta}) & \cdots & \pi(a_N|o_2;\boldsymbol{\theta}) \\ \vdots & & & \vdots \\ \pi(a_1|o_{|\mathcal{O}|};\boldsymbol{\theta}) & \pi(a_2|o_{|\mathcal{O}|};\boldsymbol{\theta}) & \cdots & \pi(a_N|o_{|\mathcal{O}|};\boldsymbol{\theta}) \end{pmatrix}. \tag{4.12}$$

The policy cannot depend on the position of the failure, otherwise we would automatically have solved the problem: the policy would suggest going in the substation in which the fault is, or in the two substations at the ends of the faulty electrical cable.

Note that what matters in (4.11) is not the absolute value of the parameters, but only the relative value of a parameter over another one: if we add a constant $c \in \mathbb{R}$ to all the parameters, there is no effect on the action probabilities, since the constant cancels out:

$$\frac{e^{\theta_{o,a}+c}}{\sum_{b\in\{v\}} e^{\theta_{o,b}+c}} = \frac{e^{c}\cdot e^{\theta_{o,a}}}{e^{c}\sum_{b\in\{v\}} e^{\theta_{o,b}}} = \frac{e^{\theta_{o,a}}}{\sum_{b\in\{v\}} e^{\theta_{o,b}}} = \pi(a \mid o(s);\boldsymbol{\theta})\,. \tag{4.13}$$

Initially, all parameters are the same and depend neither on the observation nor on the action (e.g., $\boldsymbol{\theta} = \mathbf{0}$, thus $\theta_{o,a} = 0$ for all $o \in \mathcal{O}, a \in \mathcal{A}$), so that all actions have an equal probability of being selected. This is called the *random policy*, and in our case is

$$\pi\left(a \mid o(s);\boldsymbol{\theta}\right) = \frac{e^{\theta}}{\sum_{b\in\{v\}} e^{\theta}} = \frac{e^{\theta}}{e^{\theta}\sum_{b\in\{v\}} 1} = \frac{1}{|\{v\}|}\,, \tag{4.14}$$

which means that we randomly choose a substation to visit, since they all have the same uniform probability. Actually, this is true for every choice of $\boldsymbol{\theta}$ which doesn't depend on the observation-action pair, so also any other constant $c \in \mathbb{R}$ would do (as we can see from (4.13)), but in practice, to construct a uniform policy, one usually takes $\boldsymbol{\theta} = \mathbf{0}$.

One of the advantages of using the soft-max distribution to parameterize policies is that, in the learning phase, we are able to choose a random action without introducing and manually tuning a ε term, which we would otherwise need to perform some exploration alongside the exploitation. Instead, the stochasticity is

intrinsic, since we don't perform a deterministic maximization over the actions, but we choose it according to its probability. In particular, if the optimal policy is stochastic, an argmax would never be able to approximate it, not even with a ε parameter, while the soft-max can do it by construction. On the contrary, if an action is deterministic, the soft-max can approach it as close as possible; in fact, the approximate policy can approach a deterministic policy without any problem.

If we search in this space of parametrized policies, this will give us a policy that doesn't depend on time, but only on the parameters, which we want to optimize. This means that, by construction, we have a *stationary policy*. The reason for this choice is that, in our problem, the time is not encoded like a problem of dynamic programming, in which we have a well-defined sequence of time steps. The structure of the states is already a measure of time, as the number of moves already done: the sequence of substations we already visited. So what is important is not to establish a policy at different time steps, but to create a policy with respect to the states (in our case with respect to the observables).

4.1.3 Value function and performance measure

In our problem, the equation for the *action value function* $Q_\pi(s,a)$ in (2.12), given that the state is $s = (\, x_g, o = (v_k, \{v\})\,)$, the action is $a \in \{v\}$ and the next state is $s' = (\, x_g, o' = (v_{k+1} = a, \{v'\})\,) = \sigma(s,a)$ (since the system is deterministic), becomes, thanks to (4.7) and (4.9),

$$\begin{aligned} Q_\pi(s,a) &= \sum_{s'} p(s' \mid s,a) \left[r(s,a,s') + \sum_{a'} \pi(a'|o(s');\boldsymbol{\theta}) Q_\pi(s',a') \right] \\ &= \left(d_{v_k,a} \cdot n_k + \sum_{a' \in \{v'\}} \pi\Big(a' \Big| o\big(\sigma(s,a)\big);\boldsymbol{\theta}\Big)\, Q\big(\sigma(s,a),a'\big) \right) . \end{aligned} \tag{4.15}$$

Now, let us find the formula for $\rho_0(s')$, the probability of starting in state s'. Since we have no prior information on where the fault might be, ρ_0 doesn't depend on it, so it will be uniform in x_g. The fault can happen either in one of the substations, which are $|\mathcal{C}| = N$, or on one of the electrical cables, which are $N+1$, so the number of possible x_g is $N + (N+1) = 2N+1$. In an initial

state, the current position of the technician v_k must be the dummy substation 0, which represents the position of the technician when the fault occurs, and the set of the disconnected substations must be equal to the set of all the substations $\mathcal{C}$. So ρ_0 must be 1 when the current substation is 0 and the set of the disconnected substations is equal to $\mathcal{C}$, and must be 0 for every other situation. In other words, we have that

$$\begin{aligned}\rho_0\Big(s = (\, x_g, o = (v_k, \{v\})\,)\Big) &= \Pr(x_g)\mathbb{I}(o = o_0 = (0, \mathcal{C})) \\ &= \frac{1}{2N+1}\mathbb{I}\big(v_k = 0, \{v\} = \mathcal{C}\big) \\ &= \frac{1}{2N+1}\delta_{o(s),o_0}\,,\end{aligned} \tag{4.16}$$

where o_0 is the initial observation, and in the last equivalence we used the Kronecker delta δ. Actually, according to the technicians of the AAA company, the majority of the faults happens on the electrical cables, usually when they are no longer perfectly isolated. We will try to take advantage of this information at a later moment, but for now we will suppose that every component has the same probability of being damaged.

Moreover, the formula (2.20) for the function $\eta_\pi(s')$ in our case becomes, thanks to (4.9) and (4.16),

$$\begin{aligned}\eta_\pi\Big(s' = (\, x_g, o' = (v_{k+1}, \{v'\})\,)\Big) &:= \rho_0(s') + \sum_s \eta_\pi(s) \sum_a \pi(a|o(s);\boldsymbol{\theta})p(s'|s,a) \\ &= \frac{1}{2N+1}\delta_{o',o_0} + \sum_{s\in \mathrm{pa}(s')} \eta_\pi(s)\pi(v_{k+1}|o(s);\boldsymbol{\theta})\,,\end{aligned} \tag{4.17}$$

where $\mathrm{pa}(s')$ indicates the parents of the node s' in the states' dependency graph, which represents all the possible sequences of states.

Finally, let us define the *performance measure* $J_\pi(\boldsymbol{\theta})$ as the sum of all the costs we incur, added up over time until the process is concluded. Actually, the time steps of the process are merely formal steps, since we don't keep track of the time passed, but we simply move from one substation to another. Thus, the actual physical time is in the costs, as the cost of going from one substation to another, which we measure using the time it takes to drive between them. Given

the definition of $J_\pi(\boldsymbol{\theta})$ as in (2.19), in our case we have that

$$\begin{aligned} J_\pi(\boldsymbol{\theta}) &= \sum_s \rho_0(s) \sum_a \pi_{\boldsymbol{\theta}}(a|o(s);\boldsymbol{\theta})\, Q_{\pi_{\boldsymbol{\theta}}}(s,a) \\ &= \sum_s \frac{1}{2N+1} \delta_{o(s),o_0} \sum_a \pi_{\boldsymbol{\theta}}(a|o(s);\boldsymbol{\theta})\, Q_{\pi_{\boldsymbol{\theta}}}(s,a) \\ &= \sum_{x_g} \frac{1}{2N+1} \sum_a \pi_{\boldsymbol{\theta}}(a|o_0;\boldsymbol{\theta})\, Q_{\pi_{\boldsymbol{\theta}}}\big((x_g,o_0),a\big)\,, \end{aligned} \tag{4.18}$$

where in the second equivalence we used equation (4.16), and the last equivalence derives from the fact that the sum $\sum_s \delta_{o(s),o_0}$ is equivalent to a summation over every possible position of the fault for the initial observation o_0. In particular, (x_g, o_0) represents every possible initial state for different positions of the fault.

4.1.4 Policy gradient method

Since we know every aspect of the problem, and we have a model of the environment, we can solve it via a *model-based method*. Due to the partial observability, though, we cannot use *dynamic programming* methods like the ones in [13], because we would need a policy that depends on the entire state, thus also on the hidden variable x_g. However, if we know where the fault is, the optimal solution is straightforward: if it is on an electrical cable, we need to visit the two substations at the ends of it (we might have to choose the right order); if it is in a substation, we visit it directly. Thus, we need an algorithm that can work in a POMDP.

We will use one of the *policy gradient methods* of section 2.4 in the POMDP, since they can naturally handle partial observability. In particular, we will perform a *gradient descent* — since we want to *minimize* our *cost* $J_\pi(\boldsymbol{\theta})$ — on the parameters $\boldsymbol{\theta}$ of the policy $\pi_{\boldsymbol{\theta}}$, where we said that the latter depends only on the observations, and so will do the gradient.

The formula of the gradient in (2.21) in our case becomes

$$\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta}) = \sum_{s\in\mathcal{S}} \eta_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a\in\mathcal{A}(s)} Q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|o(s);\boldsymbol{\theta})\,. \tag{4.19}$$

So, given that $\boldsymbol{\theta}$ is a matrix $\boldsymbol{\theta} = (\theta_{o,a})_{o\in\mathcal{O},a\in\mathcal{A}}$, we have that also the gradient

is a matrix:

$$\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_{o_1,a_1}} J & \cdots & \frac{\partial}{\partial \theta_{o_1,a_N}} J \\ \frac{\partial}{\partial \theta_{o_2,a_1}} J & \cdots & \frac{\partial}{\partial \theta_{o_2,a_N}} J \\ \vdots & & \\ \frac{\partial}{\partial \theta_{o_{|\mathcal{O}|},a_1}} J & \cdots & \frac{\partial}{\partial \theta_{o_{|\mathcal{O}|},a_N}} J \end{pmatrix}, \tag{4.20}$$

where $N = |\mathcal{A}| = |\mathcal{C}|$ is the number of initially disconnected substations.

First of all, let us compute the gradient of the policy π. Given the equation of the policy in (4.11), we have that its derivative is

$$\begin{aligned} \frac{\partial}{\partial \theta_{o',a'}} \pi\Big(a \;\Big|\; o = (v_k, \{v\}); \boldsymbol{\theta}\Big) &= \frac{\partial}{\partial \theta_{o',a'}} \left(\frac{e^{\theta_{o,a}}}{\sum_{b \in \{v\}} e^{\theta_{o,b}}} \right) \\ &= \delta_{o',o} \cdot \frac{\delta_{a',a} \cdot e^{\theta_{o,a}} \cdot \sum_{b \in \{v\}} e^{\theta_{o,b}} - e^{\theta_{o,a}} \cdot e^{\theta_{o,a'}}}{(\sum_{b \in \{v\}} e^{\theta_{o,b}})^2} \\ &= \delta_{o',o} \cdot \left(\frac{\delta_{a',a} \cdot e^{\theta_{o,a}} \cdot \sum_{b \in \{v\}} e^{\theta_{o,b}}}{(\sum_{b \in \{v\}} e^{\theta_{o,b}})^2} - \frac{e^{\theta_{o,a}}}{\sum_{b \in \{v\}} e^{\theta_{o,b}}} \cdot \frac{e^{\theta_{o,a'}}}{\sum_{b \in \{v\}} e^{\theta_{o,b}}} \right) \\ &= \delta_{o',o} \cdot (\delta_{a',a} \pi(a|o;\boldsymbol{\theta}) - \pi(a|o;\boldsymbol{\theta}) \pi(a'|o;\boldsymbol{\theta})) \\ &= \delta_{o',o} \left(\delta_{a',a} - \pi(a'|o;\boldsymbol{\theta})\right) \pi(a|o;\boldsymbol{\theta}), \end{aligned} \tag{4.21}$$

since if $o \neq o'$ we have that $\theta_{o',a'}$ and $\theta_{o,a}$ are ultimately different parameters. In particular, for the random policy (so when $\boldsymbol{\theta} = \mathbf{0}$) we have that

$$\frac{\partial}{\partial \theta_{o',a'}} \pi\Big(a \;\Big|\; o = (v_k, \{v\}); \boldsymbol{\theta}\Big)\Big|_{\boldsymbol{\theta}=\mathbf{0}} = \delta_{o',o} \left(\frac{1}{|\{v\}|} \delta_{a',a} - \frac{1}{|\{v\}|^2} \right) \tag{4.22}$$

The gradient of the policy is, in general, a tensor with dimensions $|\mathcal{O}| \times |\mathcal{A}| \times |\mathcal{O}| \times |\mathcal{A}|$. For a fixed observation o and for a fixed action a, however, it reduces to the following matrix:

$$\nabla_{\boldsymbol{\theta}} \pi(a|o;\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_{o_1,a_1}} \pi(a|o;\boldsymbol{\theta}) & \cdots & \frac{\partial}{\partial \theta_{o_1,a_N}} \pi(a|o;\boldsymbol{\theta}) \\ \vdots & & \vdots \\ \frac{\partial}{\partial \theta_{o_{|\mathcal{O}|},a_1}} \pi(a|o;\boldsymbol{\theta}) & \cdots & \frac{\partial}{\partial \theta_{o_{|\mathcal{O}|},a_N}} \pi(a|o;\boldsymbol{\theta}) \end{pmatrix} \tag{4.23}$$

Given this, we have that

$$\begin{aligned}\nabla_{\theta_{o',a'}} J_\pi(\boldsymbol{\theta}) &= \sum_{s\in\mathcal{S}} \eta_\pi(s) \sum_{a\in\mathcal{A}(s)} Q_\pi(s,a)\nabla_{\theta_{o',a'}}\pi(a|o(s)) \\ &= \sum_{s} \eta_\pi(s) \sum_{a} Q_\pi(s,a)\delta_{o',o(s)}\Big(\big(\delta_{a',a} - \pi(a'|o(s))\big)\,\pi(a|o(s))\Big) \\ &= \sum_{x_g} \eta_\pi\big((x_g,o')\big) \sum_{a} Q_\pi\big((x_g,o'),a\big)\big(\delta_{a,a'} - \pi(a'|o')\big)\,\pi(a|o')\,,\end{aligned} \tag{4.24}$$

where, as we did in equation (4.18), the sum $\sum_s \delta_{o',o(s)}$ is equivalent to a summation over every possible position of the fault for the specific observation o' on which we are deriving. In particular, (x_g, o') is the state with a fault at some position x_g and with observation o'.

The idea of the algorithm is the following. We start from a certain policy, for example, the random policy of equation (4.14), in which we choose randomly the substation to be visited. Then we compute the gradient for every parameter using (4.24). As we saw in (4.22), the gradients of the policy are simple computations (since we decided the parametrization of the policy, we were able to derive their formulas), while the computations for the objects Q_π and η_π have to be done indirectly: we have to solve the linear equations (4.15) and (4.17) for the current policy, which can be more or less computationally heavy, also depending on the method chosen.

Having done these steps, we have the value of the gradient for every parameter. Then we take a step in the parameters space, and we descend the gradient. We stop when the value of $J_\pi(\boldsymbol{\theta})$ doesn't change much in percentage.

In general, there is no guarantee that this is a convex problem in the parameters $\boldsymbol{\theta}$, on the contrary, we could have several minima. Therefore, we should perform different gradient descents starting with some policies different from the one which has $\boldsymbol{\theta} = \mathbf{0}$, to see if we can reach a different minimum, or if we always reach the same one. These are called random restarts. Of course, this doesn't guarantee finding the global minimum, but it is the best we can do to at least check that we are not stuck in a local minimum. For the random restarts, we chose the parameters $\boldsymbol{\theta}$ from a standard normal distribution $\mathcal{N}(0,1)$, in order to have both positive values and negative ones to start from.

If we have the intuition that there is a deterministic sequence of actions to be performed, we have that their parameters $\theta_{\cdot,a}$ tend to infinity. This is because the deterministic policies correspond to a matrix of one-hot vectors — in which we have 1 for only one action and 0 for all the other ones, which in the soft-max policy happens when the parameter associated with this action becomes much bigger than the others. If it is so, the gradient descent never stops and there could be problems of overflowing on the values of the policy, since the parameters $\boldsymbol{\theta}$ can become very large, and the exponentials of the soft-max would explode. A smart thing to do when this happens is to rewrite the parametrization in the following way (the observation o is fixed):

$$\pi(a|o) = \frac{e^{\theta_{o,a}}}{\sum_b e^{\theta_{o,b}}} = \frac{e^{-(\max_{a'} \theta_{o,a'} - \theta_{o,a})}}{\sum_b e^{-(\max_{a'} \theta_{o,a'} - \theta_{o,b})}} \,. \tag{4.25}$$

The benefit of writing the policy in this way is that, since the exponents of the two exponentials are negative (the parts in the parenthesis, $\max_{a'} \theta_{o,a'} - \theta_{o,\cdot}$, are always positive), it never explodes. In fact, negatives with large exponents "saturate" to zero rather than infinity, so we have a better chance of avoiding NaNs. This is a simple numerical trick that allows improving the stability of the algorithm.

4.1.4.1 Natural policy gradient

Following what we said in subsection 2.4.1, and given that the derivative of our policy is (4.21), we have that

$$\widetilde{\nabla}_{\theta_{o',a'}} \pi(a|o(s)) = I_{(\mathcal{O},\mathcal{A})\times(\mathcal{O},\mathcal{A})} = \delta_{o',o(s)} \delta_{a',a} \,. \tag{4.26}$$

Then the equation of the natural gradient of the performance measure $J(\boldsymbol{\theta})$ of (2.22), which was

$$\widetilde{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = F^{-1}(\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \,,$$

in our case becomes, thanks to the equation of the natural gradient of π used in

equation (4.19),

$$\begin{aligned}\widetilde{\nabla}_{\theta_{o',a'}} J_\pi(\boldsymbol{\theta}) &= \sum_{s\in\mathcal{S}} \eta_\pi(s) \sum_{a\in\mathcal{A}(s)} Q_\pi(s,a)\widetilde{\nabla}_{\theta_{o',a'}}\pi(a|o(s)) \\ &= \sum_{s} \eta_\pi(s) \sum_{a} Q_\pi(s,a)\delta_{o',o(s)}\delta_{a',a} \\ &= \sum_{x_g} \eta_\pi\big((x_g,o')\big) Q_\pi\big((x_g,o'),a'\big)\,.\end{aligned} \tag{4.27}$$

This is a very nice result, in that we have a very simple equation for the gradient of the performance measure $J(\boldsymbol{\theta})$, which is also guaranteed to converge faster to a minimum, although not a global one.

4.2 Implementation of the model

First of all, we generated all the states of the system using *backtracking*, a technique used to systematically generate all the admissible solutions of a problem (usually of combinatorial nature) [14]. It proceeds by incrementally building candidate solutions using recursion, checking at each step if a complete and valid solution is reached, and backtracking to explore other ones. In our case, the function starts from a complete initial state, so, besides the current position of the technician and the set of disconnected substations, it also requires the position of the fault, and it determines all the possible actions that can be taken from that state. Then it computes all the subsequent states, determining each time which substations can be reconnected by doing one of the possible actions, and it continues until it has reconnected all the substations. We can reconstruct all the possible states of the system by looping over all the possible positions of the fault, so on all the possible initial states. While constructing the states, the function also creates the states' dependency graph, in which, given a state, we can find all the possible states in which we can end up doing an admissible action. This is actually a forest of trees: each graph is a tree because we can never return to a previous state, and it is a forest since the states are grouped by the position of the fault: if we start in a given initial state we can never end up in a state with a different position of the fault.

From the list of states, we create the list of observables, and we associate each state to its uniquely defined observation using a dictionary.

After having done that, we proceed to construct the matrix of parameters $\boldsymbol{\theta}$, initialized to zero, and the matrix of the policy $\pi_{\boldsymbol{\theta}}$, with the same dimensions of the previous one, using the soft-max parametrization.

Then we proceed with the computation of the matrix of Q_π. We construct the matrix $R = (R[s,a])_{s\in\mathcal{S},a\in\mathcal{A}}$, which stores the immediate cost of being in state s and doing action a, which thanks to (4.7) is:

$$R[s,a] = r(s,a,s') = d_{v_k,a} \cdot \sum_{v\in\{v\}} u_v \,, \tag{4.28}$$

if action a is possible in state s, otherwise it is zero. Then we use it to initialize the matrix which will store the values of Q_π. Recall that in the equation (4.15) for Q_π, we use the values of Q_π of the next states s' to compute the value of Q_π for the current state s. Thus, we need to use the states' dependency graph from the leaves to the roots to compute the matrix of Q_π bottom-up. We perform a personalized *breadth-first-search*, or BFS, on the states' dependency graph, in order to be able to add each next-state contribution to the value of Q_π for the current state. A breadth-first-search is an algorithm for exploring a graph, which visits every node and every edge of the latter. In a breadth-first-search, the nodes are visited in order of increasing distance from the source of the visit, where the distance among the source and a generic node is the minimum number of edges in a path among them [14]. In our modified BFS, instead, we start from the terminal states in the leaves, and then we visit all the edges of the graph in a breadth-first-search order, with every edge traversed in the reverse direction in order to arrive at the initial state in the root. In particular, the major modification is that we visit all the out-edges of a node before visiting that node itself, in order to accumulate its Q_π value — by adding all the terms deriving from its children — before using it to compute the Q_π value of another node.

One might wonder why, being (4.15) a linear system, we didn't use any of the already existing algorithms to solve it. This is because we don't have the matrix of the linear system explicitly, and we don't have an immediate way to create it, but

it is required for algorithms like the Gauss elimination method (a direct method) or the Jacobi or Gauss-Seidel methods (iterative ones). So to compute it, we would need to first traverse the graph like we explained, writing the coefficients of the equations' terms of the linear system in A. Then we would still have to solve the linear system, for example with one of the previous methods. However, instead of doing these two passages, we can directly solve the system traversing the graph. This applies to the computation of η_π, as well.

After this, we proceed by computing the vector of η_π. In equation (4.17) we have that, in order to compute the value of η_π for a state s', we use the values of η_π of all its preceding states s. Thus, we will use the states' dependency graph from the roots to the leaves to compute the η_π matrix top-down. As we did for the matrix of Q_π, we perform a modified BFS on the states' dependency graph, which visits all the edges. Our BFS visits a node only after all its in-edges have been visited; this ensures that we have accumulated the node's η_π value, by having added all the terms deriving from its parents, before using it to compute the η_π value of another node.

We could not have used a depth-first search, or DFS, to compute Q_π and η_π. In a DFS, contrary to a BFS, after visiting a node the search proceeds as far as possible from it along a path, until it reaches a node whose adjacent nodes have all already been visited. Then the search goes back along the last edge and continues moving along another path not yet visited [14]. Thus, using a DFS, it would be impossible to visit all the in-edges (or the out-edges in the reverse order) of a node before visiting the node itself.

Having the matrix of the policy $\pi_{\boldsymbol{\theta}}$ and the matrix of Q_π, we can compute the performance measure $J_\pi(\boldsymbol{\theta})$ using the last equivalence of (4.18). Instead, for the computation of the gradient of $J_\pi(\boldsymbol{\theta})$, $\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$, we also use the matrix η_π, following equation (4.24). Since the convergence with the ordinary gradient was quite slow, we also implemented the natural gradient $\widetilde{\nabla}_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$, following equation (4.27). Given the higher values of $\boldsymbol{\theta}$ that the natural gradient can achieve, it is mandatory to implement the clipped version of the policy (4.25).

How to choose the learning parameter α appropriately? We want the difference

$\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k$ in (2.18) to be small; in particular, we want $\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k \ll 1$, therefore, we must have $\alpha \left|\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})\right| \ll 1$. Given that we typically have $Q_\pi \sim 10^7$ and $\eta_\pi \sim 10^{-2}$ in the expression of $\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$, we choose $\alpha \sim 10^{-7}$. In our numerical runs, we actually used an adaptive α computed as $1/\max(Q_\pi)$ at each iteration, in order to speed up the convergence and to avoid using a really small or big α with respect to the value of Q_π. This choice is still consistent with our previous argument; however, an adaptive α automatically takes into account the fact that the values of Q_π increase when we have many initially disconnected substations.

After having computed all the elements that we need, we proceed to perform the gradient descent, stopping when the relative error among two subsequent values of $J_\pi(\boldsymbol{\theta})$, normalized by α, is less than 0.01%.

The pseudocode implementing our algorithm is shown in algorithm 1.

Algorithm 1: Policy gradient descent

Data: The error tollerance: tol $= 0.0001$

Initialize $\boldsymbol{\theta} = \mathbf{0}$

Loop

- Compute the matrix $\pi \leftarrow \left(\pi(a|o(s);\boldsymbol{\theta})\right)_{o\in\mathcal{O},a\in\mathcal{A}}$
- Compute the vector $\eta \leftarrow \left(\eta_\pi(s)\right)_{s\in\mathcal{S}}$
- Compute the matrix $Q \leftarrow \left(Q_\pi(s,a)\right)_{s\in\mathcal{S},a\in\mathcal{A}(s)}$
- $\alpha \leftarrow 1/\max(Q)$
- Compute $G \leftarrow \nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$ or $G \leftarrow \widetilde{\nabla}_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$
- Compute $J \leftarrow J_\pi(\boldsymbol{\theta})$
- error $\leftarrow |J - J_{\text{old}}|/J_{\text{old}} \cdot 1/\alpha$
- **if** error $<$ tol **then**
  - **break**
- **end**
- Compute $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha G$
- Set $J_{\text{old}} = J$

end

Chapter 5

Results

In this chapter, we will comment on the results obtained with the algorithms we described in the previous chapters: we will both analyze them individually and compare them with each other.

We first implemented the ordinary policy gradient descent (PG), using the formula for the gradient $\nabla_{\boldsymbol{\theta}} J_\pi(\boldsymbol{\theta})$ in equation (4.24). Then, realizing how slow the convergence was, we decided to implement also the natural policy gradient algorithm (NPG), following equation (4.27). While reaching the same policy (the difference is in the order of 10^{-2}), the NPG reaches convergence incredibly faster than the PG algorithm, as we can see from the right panel of Figure 5.1. The figure reports the convergence times of the two algorithms, for different numbers of initially disconnected substations. From the left panel, which reports the same times but in logarithmic scale, we can see that both the two curves are exponentials, with approximately the same rate, but the NPG one has an initial value about 10^{-2} times smaller than the PG one, which considerably reduces the computation time. For example, with 13 initially disconnected substations, the convergence of the PG took 20 hours, while the convergence of the NPG took 17 minutes.

These two algorithms find a policy that is not completely deterministic: in fact, some states have a stochastic policy over their possible actions. This means that we are on a side of the simplex which represents the policy space, and not

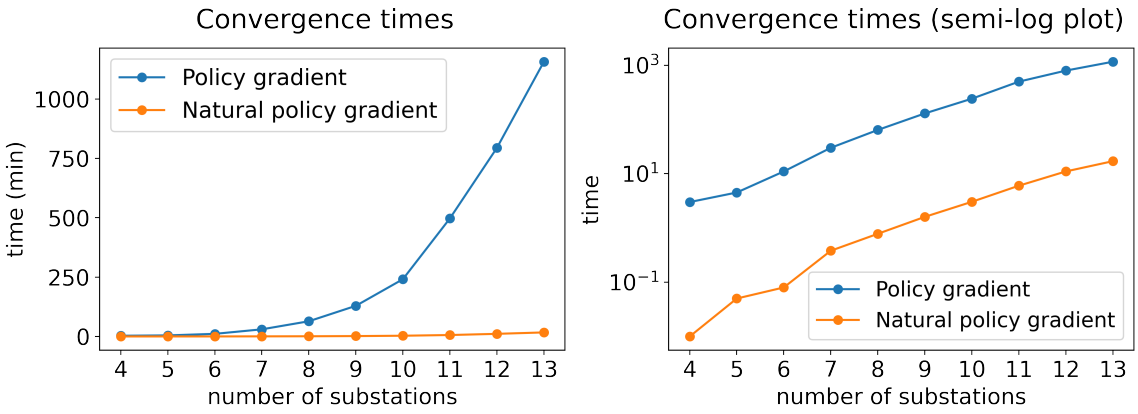


Figure 5.1: Comparison of the convergence times of PG and NPG algorithms for different numbers of initially disconnected substations. The second figure shows the same times but in logarithmic scale.

in a corner. However, when, given a fault, we followed the actions dictated by the policy, we ended up only in states with a deterministic choice of the action to be performed. In particular, an interesting thing to notice is that, in the initial state, the policy always prescribes to go to the substation in the middle, like the bisection algorithm. Then, for subsequent substations, the two policies begin to differ, giving very different results, as we will see later.

The first thing to look at in a policy gradient algorithm is whether the gradient descent reached convergence. The first, immediate, check is to see if the error is decreasing. We checked this both for the PG algorithm and for the NPG algorithm, and both of them were decreasing, as we can see from the plots of Figure 5.2. In the left panel we have the errors of the ordinary policy gradient with both axes in logarithmic scale. Since we have two straight segments, the errors follow two different decreasing power laws; until just before 10^2 steps, the errors decrease slowly, while afterwards they decrease faster. Instead, in the right panel we can see the errors of the natural policy gradient: here we have two lines with different slopes, which indicates that initially the errors decrease faster, while after about 100 steps they start decreasing according to an exponential with a lower rate.

However, this is not enough to say that the algorithm converged, this is a mere

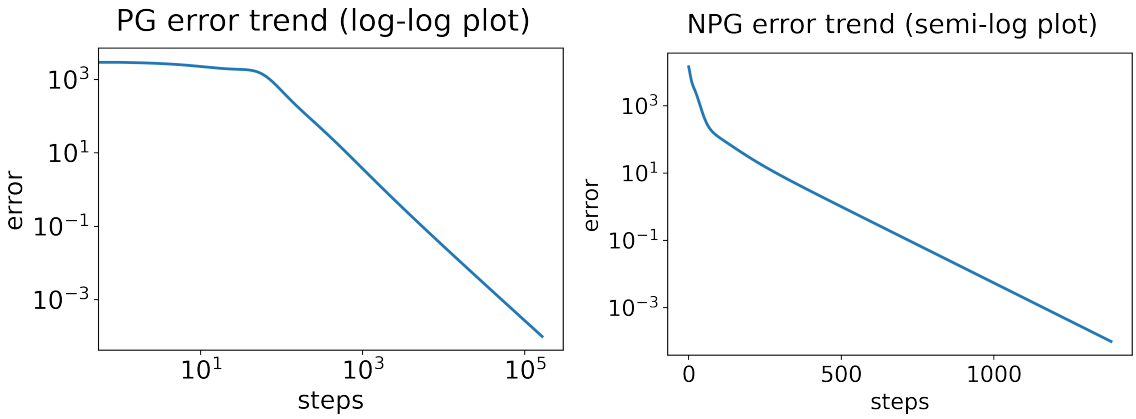


Figure 5.2: On the left there are the PG errors, while on the right the NPG ones, both for 5 initially disconnected substations. They are both decreasing, so the gradient descent is following a minimum, as we want it to do.

indicator that the gradient descent is indeed following the gradient and moving towards a minimum. Thus, what we did next was to examine the trajectories of the parameters $\boldsymbol{\theta}$ in the parameters space and the trajectories of the policies $\pi_{\boldsymbol{\theta}}$ in the policy space. In Figure 5.3 and Figure 5.4 we can see these two quantities for the PG, where we selected a specific episode with 5 initially disconnected substation and the fault among the second and the third substation. In Figure 5.5 and Figure 5.6 we can see these two quantities for the NPG for the same exact episode. In each panel, the current state is reported in the title, and the plot represents the trajectory of the parameters $\boldsymbol{\theta}$ or the policy $\pi_{\boldsymbol{\theta}}$ for that specific state. Thus, we can see the parameters or the value of the policy for each specific action that can be taken in that state. The numbers in the states and in the labels represent the codes of the substations, and the faulty cable is indicated with the codes of the two substations at its ends. The subsequent panels represent the next states in which we end up taking the action suggested by the policy in the previous panel.

The most important thing to notice in these figures is that all the trajectories are continuous, without sudden turns backward. Additionally, in the PG trajectories of $\boldsymbol{\theta}$ we can see that one of the parameters grows to infinity, while the others either decrease to infinity or are zero (the latter correspond to actions that cannot

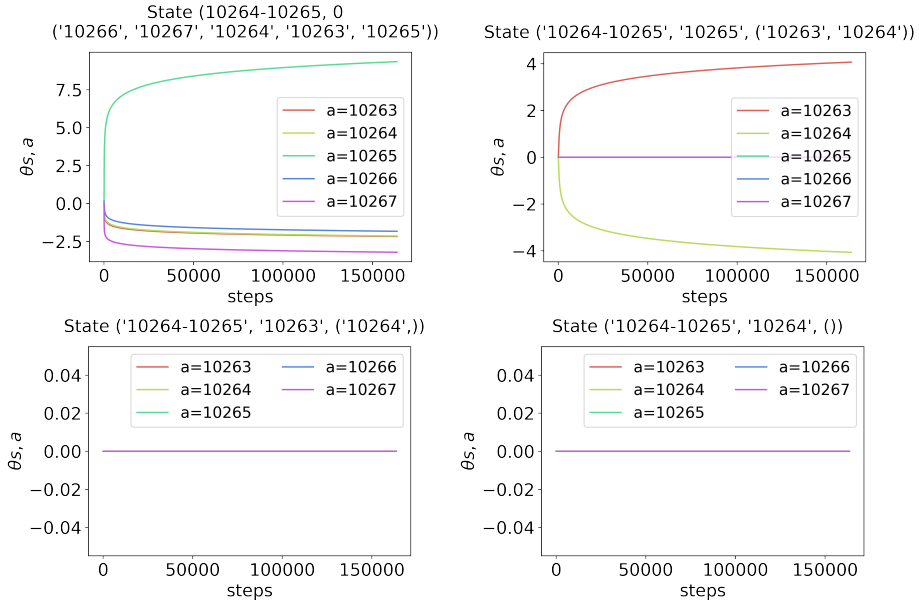


Figure 5.3: Trajectories of the parameters $\boldsymbol{\theta}$ in the parameters space for PG, for a specific episode (which starts in the top-left corner and goes from left to right, line by line) with 5 initially disconnected substation.

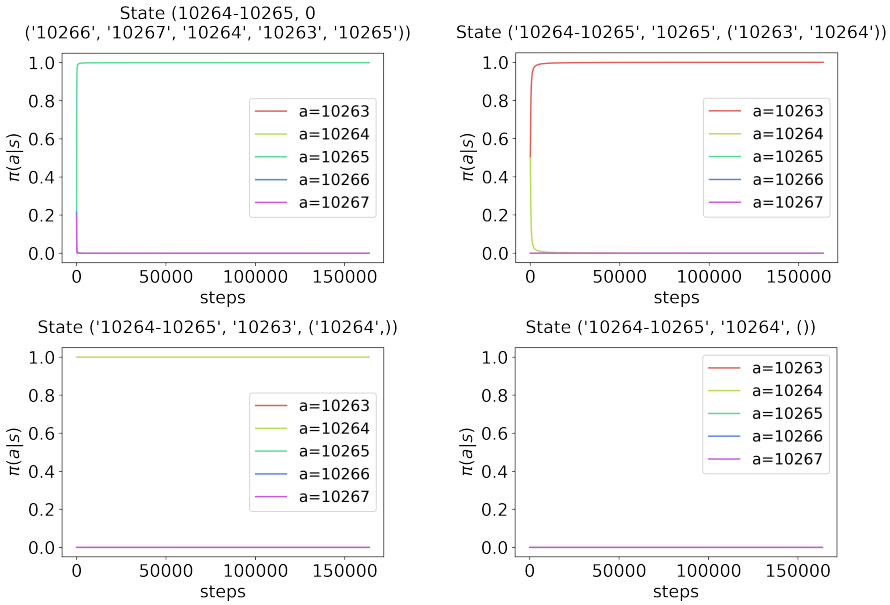


Figure 5.4: Trajectories of the policies $\pi_{\boldsymbol{\theta}}$ in the policy space for PG, in the exact same previous setting.

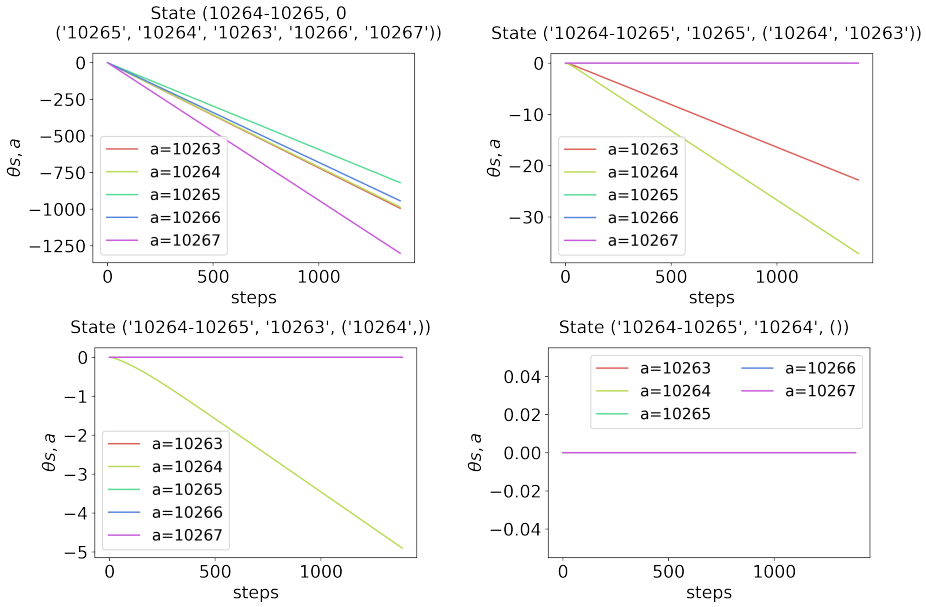


Figure 5.5: Trajectories of the parameters $\boldsymbol{\theta}$ in the parameters space for NPG, for a specific episode (which starts in the top-left corner and goes from left to right, line by line) with 5 initially disconnected substation.

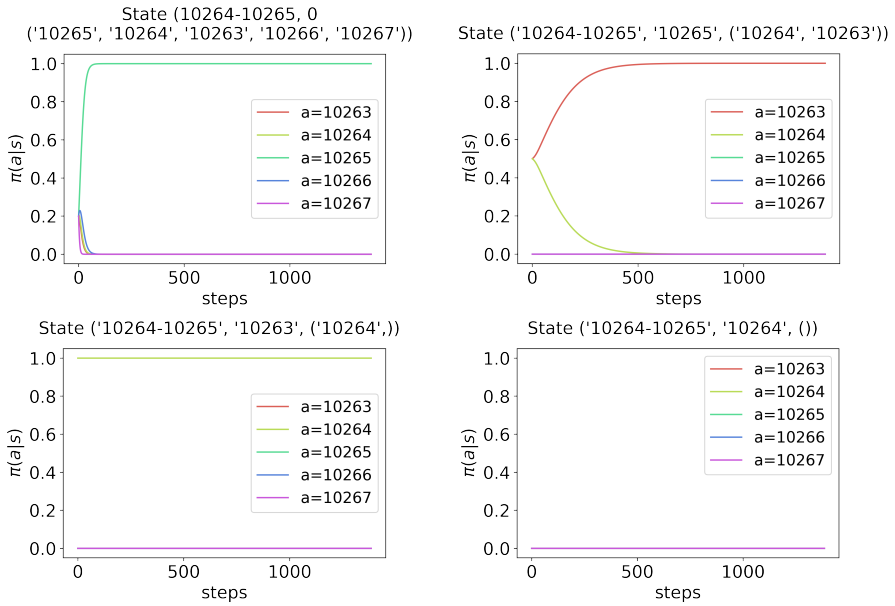


Figure 5.6: Trajectories of the policies $\pi_{\boldsymbol{\theta}}$ in the policy space for NPG, in the exact same previous setting.

be taken in that state). In particular, in the last panel all parameters are zero, since in the terminal state we cannot take any action. Moreover, in the second to last panel we have that again all θs are zero, because we only have one possible action, so the corresponding θ is zero (in order for the soft-max policy to be equal to 1), while the others are zero because they correspond to inadmissible actions. Instead, in the NPG we have that all the parameters $\boldsymbol{\theta}$ decrease, since we are not following the steepest direction in the parameters space, but the steepest direction with respect to the Fisher metric. Even with rather different values of the parameters $\boldsymbol{\theta}$, the policies are the same (as we already said, when compared among them, the difference is in the order of 10^{-2}), and from the figures we can see that they reached convergence.

Instead, the standard strategy of checking whether the gradient reached a zero is not applicable in this case: in fact, since we find a stochastic policy, we are not in a corner of the space, in which the gradient is exactly zero, but we are on a side, so it can have values different from zero. Indeed, this is what happens if we look at the gradient matrix: for some states it is not equal to zero, even significantly.

Another check we performed to assess the convergence is to initialize the NPG algorithm with the parameters $\boldsymbol{\theta}$ found by the PG algorithm, and vice versa. What we got is that the NPG algorithm continues the gradient descent for about another 600 steps, reducing a bit the value of $J_\pi(\boldsymbol{\theta})$. Actually, the difference is in the order of 10^{-4}, which is around the 0.001%, so it is negligible. Instead, the PG algorithm exits immediately the gradient descent loop, as soon as the check on the error is performed, and the values of $J_\pi(\boldsymbol{\theta})$ are identical (with a difference of 10^{-10}), as well as the policy (with a difference of 10^{-15}).

In order to check whether we reached a global minimum or a local one, since we face a non-convex problem, we also performed some random restarts, with the parameters $\boldsymbol{\theta}$ initialized from a standard normal distribution $\mathcal{N}(0, 1)$. At each run, the algorithm converges to the exact same policy, so we are quite confident that we reached a global minimum. Refs. [15, 16] show that, under some specific conditions, the policy gradient performance measure $J_\pi(\boldsymbol{\theta})$ has a unique global minimum, even if it is non-convex. Indeed, they rule out the possibility that the

performance measure has another minimum that can be reached from a different initial condition. Given our results, our model could be in this class of problems; however, we did not explore this route, as it was beyond the scope of this thesis.

Since we computed the policy matrix for all the various bisection algorithms, we were able to compute the exact value of $J_\pi(\boldsymbol{\theta})$ for all of them, and we were able to make an effective comparison among all the algorithms. In Figure 5.7, we can see the values of $J_\pi(\boldsymbol{\theta})$ computed by the various algorithms for different numbers of initially disconnected substations. We selected an electrical line of Trieste's power grid, and while keeping the initial position of the technician and the last remotely controlled substation fixed, we fictitiously moved the first remotely controlled substation by one substation each time, in order to incrementally increase the number of initially disconnected substations by one. We can clearly see that the policy gradient algorithms are the better ones, with the NPG a little better with higher numbers of disconnected substations. Instead, the different bisection algorithms perform almost the same, with the weighted bisection a bit worse than the others.

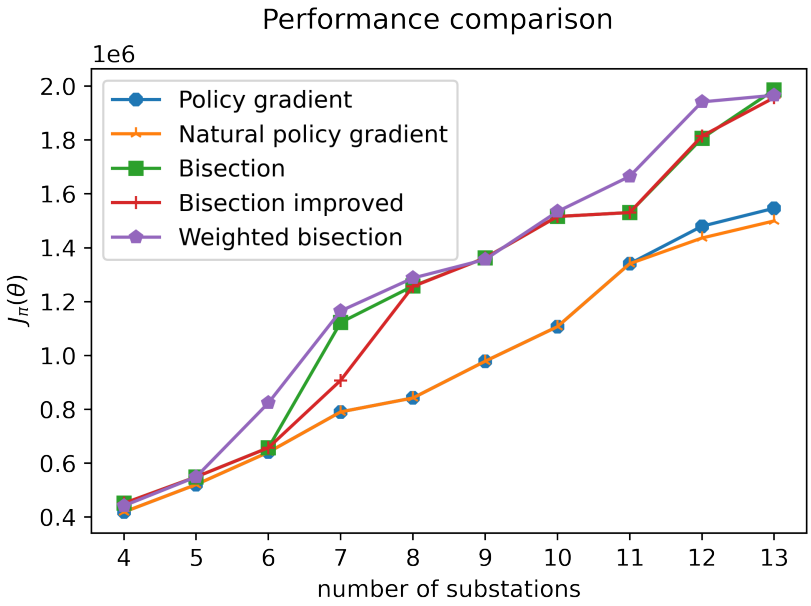


Figure 5.7: Comparison of the values of the performance measure $J_\pi(\boldsymbol{\theta})$ among the various algorithms for different numbers of initially disconnected substations.

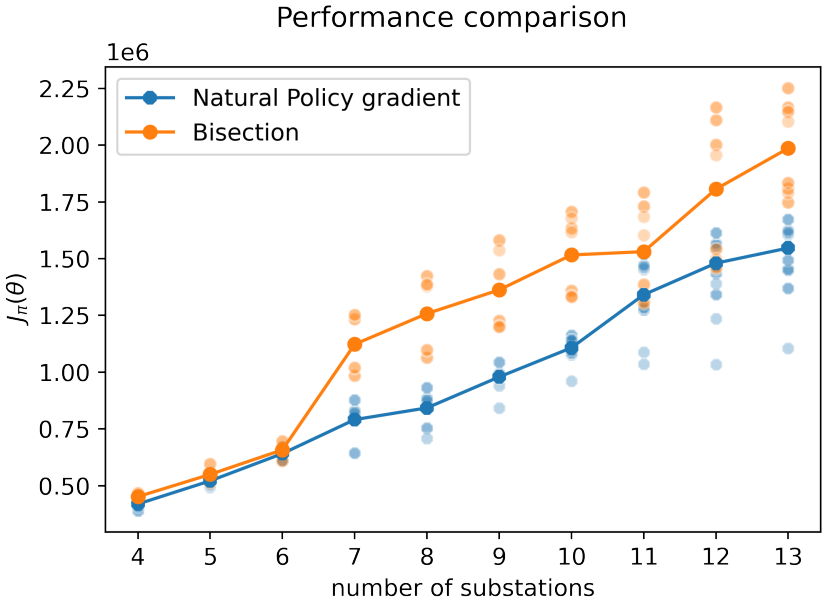


Figure 5.8: Comparison of the values of the performance measure $J_\pi(\boldsymbol{\theta})$ between NPG and bisection (darker dots), along with the values for every specific position of the fault (lighter dots), for different numbers of initially disconnected substations.

In Figure 5.8 we can see the two lines of the NPG and bisection algorithms for $J_\pi(\boldsymbol{\theta})$ of the previous plot (the darker dots), with an addition. In fact, we also plotted the performance measure $J_\pi(\boldsymbol{\theta})$ computed for a specific position of the fault, meaning that we allowed x_g to take only one specific position at a time, and we iterated over every possible value of that variable and over different numbers of initially disconnected substations (the lighter dots). We considered only the faults on the electrical cables, otherwise there would have been too many points, and since according to the data collected by the technicians they are the most frequent ones, they are the most representative. We can see that, with 11 and 12 initially disconnected substations, for some positions of the fault, the bisection algorithm is actually a slightly better than the NPG one, as some orange points are under some blue points. We notice a slight overlap with 5 and 6 substations as well, but it is not particularly significant. Instead, for the other numbers of initially disconnected substations, the policy gradient is always better than the bisection. In any case, what is important is that *on average* the NPG is always better, as the

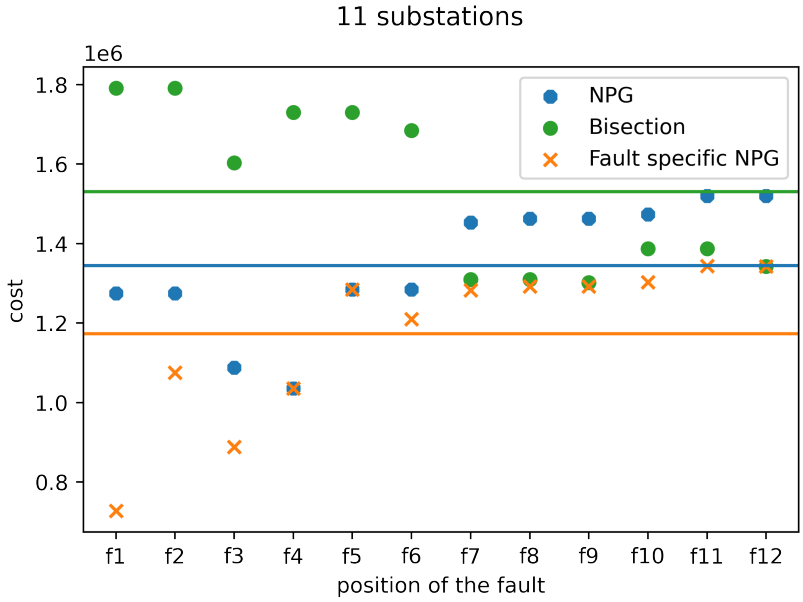


Figure 5.9: Costs for different positions of the fault in a scenario having 11 initially disconnected substations. The horizontal lines represent the average cost.

value of $J_\pi(\boldsymbol{\theta})$ is lower, which was our goal.

In Figure 5.9 we can see a detail of the previous plot for 11 initially disconnected substations. We computed the costs of the fault for different positions of the fault of the NPG, the bisection algorithm and an NPG ran only with that specific position for the fault. Even here we considered only the faults on the electrical cables, and `f1` indicates the fault between the first remotely controlled substation and the first disconnected substation, `f2` indicates the fault between the first and the second disconnected substations, and so on so forth. We can notice that, if we perform a gradient descent with only a possible position of the fault, the NPG is able to optimize for it, and finds a lower value for the cost than the general NPG. However, we have to remember that the fault specific NPG assumes that we know the position of the fault, so it is not applicable in our case, since it has a policy for every position of the fault, and not a general one that we can use when we don't know the position of the fault.

Chapter 6

Conclusions

We have succeeded in developing a method that reaches, on average, a better solution than the current one used by the technicians. However, it is possible that, using a different cost structure, these results do not remain the same. The performances of the heuristic technique and the policy gradient algorithm may increase or decrease if we change the definition of the cost. Nevertheless, it is remarkable that, even in this case, our algorithm would continue to be applicable and scalable. In fact, besides our specific results, what is important is to have developed a method that can systematically find an optimal policy in a model-based context, in which we are, since we know the consequences of our actions.

Another possible approach would be to use a model of *mixed integer (linear) programming*, where only some unknown variables are required to be integers, while the others are continuous. In this case, it is possible to find an upper bound on the cost of the restoration process. This is done by putting ourselves in the worst case, in which we have to visit *every* initially disconnected substation, and compute the cost for every possible sequence of visits to the substations and for every possible position of the fault. Then, for every possible visiting path, we take the maximum cost among the different positions of the fault. As an estimate for the cost of the fault, we take the minimum cost among the ones corresponding to the different visiting sequences. In this way we can say that, for sure, we will never

pay more than what we estimated, but very likely we will pay a lower price, since sometimes we will be able to reconnect more substations at once. An algorithm has been developed from third parties as a spin-off of this work, and could be expanded in future works.

In real life, though, the costs are not quite like we modeled them, they are much more random, since they depend on many factors: whether it is day or night, whether the instrumental test is applicable, the weather conditions, and, of course, many other unexpected events can happen. If we are to model more accurately these costs, first of all we would have to collect data on them, using maybe a Telegram bot connected to an AWS server, with which the technicians can interact while solving the fault. Then a whole other class of algorithms would be needed to solve this problem: the *model-free* ones, specifically designed to handle these uncertainties about the costs. This would be an interesting direction for future works, despite we will not explore it in this thesis.

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts (MA): MIT Press, 2018. URL: http://incompleteideas.net/book/the-book.html (cit. on pp. 11, 12, 18–20, 38).

[2] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State of the Art*. Springer Verlag, 2012. ISBN: ISBN: 978-3-642-27645-3 (cit. on p. 12).

[3] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial Intelligence* 101.1 (1998), pp. 99–134. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(98)00023-X (cit. on pp. 13, 16, 17).

[4] William Uther. "Markov Decision Processes". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 642–646. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_512 (cit. on p. 13).

[5] Matthijs T. J. Spaan. "Partially Observable Markov Decision Processes". In: *Reinforcement Learning: State of the Art*. Ed. by Marco Wiering and Martijn van Otterlo. Springer Verlag, 2012, pp. 387–414 (cit. on pp. 15, 17).

[6] Pascal Poupart. "Partially Observable Markov Decision Processes". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 776–776. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_642 (cit. on pp. 16, 17).

[7] D. Aberdeen and Jonathan Baxter. "Scaling Internal-State Policy-Gradient Methods for POMDPs". In: *International Conference on Machine Learning*. 2002 (cit. on p. 18).

[8] Jan Peters and J. Andrew Bagnell. "Policy Gradient Methods". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 774–776. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_640 (cit. on pp. 18, 19).

[9] Richard S Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 2000 (cit. on p. 19).

[10] Shun-ichi Amari. "Natural Gradient Works Efficiently in Learning". In: *Neural Computation* 10.2 (Feb. 1998), pp. 251–276. ISSN: 0899-7667. DOI: 10.1162/089976698300017746 (cit. on p. 20).

[11] Jan Peters and Stefan Schaal. "Natural Actor-Critic". In: *Neurocomputing* 71.7 (2008), pp. 1180–1190. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2007.11.026 (cit. on p. 21).

[12] Daniel Hennes et al. "Neural Replicator Dynamics: Multiagent Learning via Hedging Policy Gradients". In: *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '20. Auckland, New Zealand: International Foundation for Autonomous Agents and Multiagent Systems, 2020, pp. 492–501. ISBN: 9781450375184 (cit. on p. 22).

[13] Richard E. Bellman. *Dynamic Programming*. Princeton Landmarks in Mathematics and Physics. Princeton University Press, 1957. ISBN: 0-691-07951-X (cit. on p. 44).

[14] Alan Bertossi and Alberto Montresor. *Algoritmi e strutture dati*. Terza edizione. CittàStudi Edizioni, 2014. ISBN: 9788825173956 (cit. on pp. 48–50).

[15] Jalaj Bhandari and Daniel Russo. "Global Optimality Guarantees For Policy Gradient Methods". In: *CoRR* abs/1906.01786 (2019). arXiv: 1709.05584 (cit. on p. 58).

[16] Jalaj Bhandari and Daniel Russo. *A Note on the Linear Convergence of Policy Gradient Methods*. 2020. arXiv: 2007.11120 (cit. on p. 58).

Index