Planning problems with rechargable energy in non-deterministic environments

Arthur Al-Sett
Supervisor: Dr Sebastian Junges
Second supervisor: Dr Wieb Bosma

Radboud University

A thesis submitted in partial fulfilment
of the requirements for the degree of
Bachelor in Mathematics

August 2023

Contents

1 Introduction

Consider e.g. a drone that needs to deliver parcels. The locations of the drone are modelled as vertices of a graph, referred to as *states* in this thesis. The drone can go from state to state by taking *actions*, which typically represent movement. We assume that when the drone takes an action, it is uncertain what its next state will be. To be more precise, the next state is chosen according to a probability distribution. In particular, different states can have different probabilities for being the drone's next state. In practice the uncertainty of the outcome of an action could be caused by interference from environmental factors such as the wind. For example, if a drone flies north for ten seconds with a fixed amount of wattage, the actual distance flown would vary depending on the velocity of the wind.

We want the drone to reach a certain special state, called a *target state*. In this example that would be a location where a parcel needs to be delivered. Furthermore, we assume that the drone's energy supply is limited and that it can be recharged in a subset of the states: the *reload states*.

Instead of using a concrete example such as drones, the generic term *agent* will be used to refer to entities that are being controlled. In this thesis we aim to use algorithms to solve the planning problem where an agent needs to reach a target state without running out of energy. *Strategies* will be defined as functions that decide at any point in time what the agent's next action should be, given the information of which states it has visited and which actions it has taken so far. Essentially, strategies decide how the agent should be controlled. The goal is to use algorithms to create a strategy that solves the aforementioned planning problem.

The majority of this thesis is based on [3]. In particular, most of the definitions, statements, and proofs are based on those from [3], but we have tried to mention it explicitly wherever they are copied verbatim.

2 Preliminaries

We let zero be included in $\mathbb{N}$ and define $\overline{\mathbb{N}} := \mathbb{N} \cup \{\infty\}$. Vector indices are often written as subscripts. However, we will sometimes use subscripts for vectors to denote something other than indices. So we use the notation $vector(index)$ instead of $vector_{index}$.

We assume the reader has some experience with probability theory. A *probability distribution* over an at-most-countable set X is a function $\Pr\colon X \to \mathbb{R}$ such that $\sum_{x \in X} \Pr(x) = 1$. $\mathcal{D}(X)$ denotes the set of all probability distributions on X. Now we introduce the model:

Definition 1.
A *consumption Markov decision process* (CMDP) is a tuple $\mathcal{M} = (S, A, \Delta, C, R, cap)$, where

- S is a finite set of states,
- A is a finite set of actions,
- Δ is a total *transition* function $S \times A \to \mathcal{D}(S)$,
- C is a total *consumption* function $S \times A \to \mathbb{N}$,
- $R \subseteq S$ is a set of *reload* states, and

- $cap \in \mathbb{N}$ is an integer, sometimes written as $cap(\mathcal{M})$ for emphasis. ◇

The states typically represent possible locations for the agent, but additional information can be encoded into them, such as whether the doors of a driverless car are open or closed. In that case there would be two states for each location. Actions are what the agent can do. Examples include driving forward, flying up, and opening its doors. The probability distribution $\Delta(s, a)$ indicates for each state s' how likely it is that the agent will visit s' after taking action a in state s. This is how we model the outcome of actions potentially[1] being uncertain due to influence of the agent's environment, e.g. the weather or other vehicles. $C(s, a)$ is the cost of taking action a in state s: the amount of resource that the agent has to spend. When the agent visits a reload state, its resource reserve is fully replenished so that the level is equal to cap: the maximum capacity of its resource reserve.

If the agent is currently in state $s \in S$ and takes action $a \in A$, it cannot reach states which are assigned a chance of zero. The only possible successor states are the states which have a non-zero chance according to the probability distribution $\Delta(s, a)$. The set of potential successors is denoted by $Succ(s, a) = \{t \in S \mid \Delta(s, a)(t) > 0\}$.

A *path* $\alpha = s_1 a_1 s_2 a_2 s_3 \ldots$ is a sequence that starts with a state, contains actions and states alternatingly, and is either infinite or ends with a state. We also require that there are no impossible transitions. That is, for all i, $s_{i+1} \in Succ(s_i, a_i)$. Let X^ω be the set of infinite sequences with elements in X, for an arbitrary set X. Furthermore, let X^* be the set of sequences in X of arbitrary but finite length. Then formally, a path is an element of $(S \times A)^\omega \cup (S \times (A \times S)^*)$ with the above constraint of having no impossible transitions.

Indices are used to refer to states in a path: α_i is defined as s_i. Furthermore, the finite prefix $s_1 a_1 s_2 a_2 \ldots s_{i-1} a_{i-1} s_i$ is denoted by $\alpha_{\ldots i}$. An infinite path is called a *run* and $Runs$ is the set of all runs in $\mathcal{M}$. Similarly, a finite path is a *history* and $Hist$ is the set of all histories in $\mathcal{M}$. If α is a history, we write $last(\alpha)$ for the last state of α.

We assume there are no cycles in the CMDPs which have zero cost. This is a realistic assumption because in practice even idling or rebooting costs energy. Furthermore, this assumption allows the results to be presented in a clearer way with fewer exceptions.

2.1 Strategies

The behaviour of the agent is controlled by a *strategy*: a function $Hist \to A$ which decides which action should be taken, given a history. Strategies are what we wish to generate. Several requirements can be set for strategies.

When the agent acts according to a strategy σ, the resulting run is non-deterministic because the results of actions are in general non-deterministic. Therefore we cannot write "*the* run resulting from acting according to σ," but we can find a *sample* run, which is just a run that could result from playing by σ.

Now, such a sample run ρ can be obtained as follows. Suppose the initial state is s_1. At first the path describing what the agent has done, is just $\alpha = s_1$. For the i-th iteration of the following process, suppose the current state of the agent is $s_i = last(\alpha)$. The strategy may need to know the whole history, so we get the next action by taking

[1] If according to $\Delta(s, a)$ a certain state has probability 1 and all other states (necessarily) have probability 0, then the outcome of taking action a in s is certain.

$a_i = \sigma(\alpha)$. The state s_{i+1} which the agent visits next, is chosen randomly according to the probability distribution $\Delta(s_i, a_i)$. Repeat the above process with $\alpha a_i s_{i+1}$, then with $\alpha a_i s_{i+1} a_{i+1} s_{i+2}$, etc. Doing this infinitely many times gives a sample run ρ.

A run is *σ-compatible* if it can be produced using the above process. That is, each action in the run corresponds to what the strategy σ would have chosen, and the chance is non-zero for each transition between states. Furthermore, a run is *s-initiated* if s is its first state. We write $Comp(\sigma, s)$ for the set of all σ-compatible, s-initiated runs. Furthermore, $\mathbb{P}^{\sigma}_{s}(X)$ is the chance that a run from $Comp(\sigma, s)$ belongs to a given measurable set X of runs. For details on $\mathbb{P}^{\sigma}_{s}(\cdot)$ and measurable sets of runs, we refer to [1].

2.2 Objectives

Clearly it would be a problem if the agent ran out of energy. In order to prevent this, we first need to track the energy level. The energy level decreases when the agent takes actions, but it is restored in reload states.[2] Therefore it is not sufficient to only sum the costs of the actions taken. However, the latter concept will be used later and is thus defined here. For a finite path α we define the *consumption* of α as

$$cons(\alpha) := \sum_{i=1}^{n-1} C(s_i, a_i)$$
$$\text{where } \alpha = s_1 a_1 s_2 a_2 \ldots s_{n-1} a_{n-1} s_n.$$

We introduce the following definition to track the actual energy level, taking reload states into account.

Definition 2. We assume a CMDP $\mathcal{M}$ and use the notation from Definition 1. Let α be a history and let the integer d ($0 \leq d \leq cap$) be the initial resource level. Then *the resource level after α, initialised by d,* written as $RL_d(\alpha)$ is defined recursively as follows.

For a history consisting of only one state ($\alpha = s_1$), $RL_d(\alpha) := d$, since the agent has taken no action and thus has consumed no energy.

If alpha contains at least one action ($\alpha = \beta a s$), then let c be the cost of the last action: $c := C(last(\beta), a)$. In the below definition, which was copied almost verbatim from [3], the symbol $\perp$ represents an insufficient resource level.

$$RL_d(\alpha) := \begin{cases} RL_d(\beta) - c & \text{if } last(\beta) \notin R \text{ and } c \leq RL_d(\beta) \neq \perp \\ cap - c & \text{if } last(\beta) \in R \text{ and } c \leq cap \text{ and } RL_d(\beta) \neq \perp \\ \perp & \text{otherwise} \end{cases} \tag{1}$$

As can be seen above, the cost of the last action is subtracted from what the agent's resource level was when it exited the previous state. If that cost is greater than that resource level, the new resource level is $\perp$. ♢

Remark 1. With the way the resource level is defined in Definition 2, it need not be a problem for the agent to start with zero initial energy assuming it starts in a reload state. Granted we have $c \leq cap$ with c as in Definition 2, the resource level will be $cap - c$ in the second state of the run. ♢

[2] In our model the resource is technically restored when the agent *leaves* a reload state.

A run ρ is *d-safe* if the resource level, initialised by d, is non-negative throughout the run, i.e. non-negative after each finite prefix of ρ. Expressed differently, using Definition 2, ρ is d-safe if $RL_d(\rho_{...i}) \neq \bot$ for all $i \geq 1$.

We define $Reach_T^i := \{\rho \in Runs \mid \rho_i \in T\}$ for all $i \geq 1$ to be the set of runs reaching a state in T at the i-th step. Furthermore, $Reach_T := \bigcup_{i \geq 1} Reach_T^i$ is the set of runs that reach some state in T at some point.

A strategy σ is *d-safe in a state s* if every run in $Comp(\sigma, s)$ is d-safe. This means that regardless of the state-transitions the agent happens to make (as this is determined by chance), it never runs out of energy. Because of the restriction from Definition 2 that $d \leq cap$, the above definition of d-safe is not valid for $d > cap$. However, we declare that all strategies are ∞-safe in each state. We say that σ is T-positive d-safe in s if it is d-safe in s and if it satisfies $\mathbb{P}_s^\sigma(Reach_T) > 0$. The latter means that there exists a run in $Comp(\sigma, s)$ that is contained in $Reach_T$. In other words, if the agent starts in s and makes choices according to σ, then the chance that the agent reaches a target state in T at some point, is not zero.

2.3 Problem statement

We introduce two vectors in order to concisely state the problems that will be solved. The vectors $Safe, SafePR_T \in \overline{\mathbb{N}}^{|S|}$ contain for each state $s \in S$ the minimal resource level $d \leq cap$ such that there exists a strategy that is d-safe in s and a strategy that is T-positive d-safe in s, respectively and ∞ if no such $d \leq cap$ exists. The "PR" in $SafePR_T$ stands for "positive reachability."

We will consider the following two problems:

- The *safety* problem:
  Given a CMDP with states S, compute the vector $Safe =: \boldsymbol{x}$ and a strategy that is $\boldsymbol{x}(s)$-safe in each $s \in S$.
- The *positive reachability* problem:
  Given a CMDP with states S and a set of target states T, compute the vector $SafePR_T =: \boldsymbol{x}$ and a strategy that is T-positive $\boldsymbol{x}(s)$-safe in each $s \in S$.

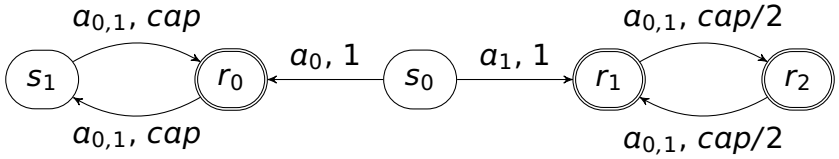
3 Safety

In this section we will solve the safety problem for a CMDP $\mathcal{M}$ which we fix for the rest of this section. To solve the safety problem we essentially need to prevent resource exhaustion. To do this, the agent must eventually reach a reload state, since there are no cycles of zero cost. Once it has reached a reload state, the agent must keep visiting reload states without running out of energy in between reload states. However, once the agent has reached a reload state, the initial resource level no longer matters, since the agent's current resource level will be cap regardless of what the initial level was. At that point, what matters is that the strategy controlling the agent only tries to guide it to reload states from which reaching another reload state can be guaranteed.

Hence, the minimal initial amount of resource for which safety can be guaranteed[3] is the minimal initial amount for which it can be guaranteed that the agent reaches

[3]Formally: the minimal amount d of initial resource for which there exists a d-safe strategy.

Figure 1: an example CMDP. The r_i are reload states. The labels for the transitions indicate actions and cost respectively. "$a_{0,1}$" means the cost and probability distribution are the same for both actions: a_0 and a_1. Transitions with probability 0 are omitted. r_0 is a "bad" reload state because the agent cannot be guaranteed to reach another reload state from there, using at most *cap* units of resource. The set of reload states for which this *can* be guaranteed, is $R' = \{r_1, r_2\}$.



a "good" reload state. Here, "good" means that it belongs to a certain set of reload states R' such that if the agent is in one of those, it can be guaranteed that the agent reaches another reload state in R'. See Figure 1. Both for finding the aforementioned minimal initial level and for finding the set R', we need a way to find the minimal initial resource level to guarantee that the agent reaches a state from a given set. This leads us to first define the vector *MinInitCons* below and to subsequently use Algorithm 1 to compute it. Then in Subsection 3.3 we solve the safety problem.

3.1 Minimum cost to reach a target state

First we introduce a number of concepts that relate to the minimum amount of resource that is required to reach a certain state. Below, a generic set $T \subseteq S$ of target states is used. This will typically be a set of reload states, but it will not always be the original set of reload states. Therefore it is clearer to have a more general definition.

Definition 3. Let $\alpha = s_1 a_1 s_2 \ldots$ be a finite or infinite path. $ReachCons_{\mathcal{M},T}(\alpha)$ is defined as the amount of the resource that is consumed in α before reaching a target state from T. If α never reaches a state in T then $ReachCons_{\mathcal{M},T}(\alpha) := \infty$. Otherwise, let $i \geq 1$ be the smallest index such that $s_i \in T$. Then $ReachCons_{\mathcal{M},T}(\alpha) := cons(\alpha_{\ldots i})$.

Furthermore, we define for a strategy σ and a state s

$$ReachCons_{\mathcal{M},T}(\sigma, s) := \sup_{\rho \in Comp(\sigma,s)} ReachCons_{\mathcal{M},T}(\rho). \qquad \diamond$$

If the agent starts in s and makes decisions according to a strategy σ, producing a sample run ρ, then the amount of resource required to reach a target state will be at most $ReachCons_{\mathcal{M},T}(\sigma, s)$. The latter is the minimal amount that is sufficient for each sample run that starts in s, since we take the supremum over the exact amounts required for all possible sample run.

Definition 4. We define the vector $MinReach_T$ such that for each state s,

$$MinReach_T(s) := \inf\{ReachCons_T(\sigma, s) \mid \sigma \text{ is a strategy}\}. \qquad \diamond$$

For each $s \in S$ we take the infimum over all strategies and for each strategy we consider the maximal amount of resource that the agent will consume before reaching

a target state when starting in s. Hence for each $s \in S$, $MinReach_T(s)$ is the minimal amount d of resource for which there is some strategy that will guide the agent to a target state if it starts in s, such that the agent consumes no more than d units of resource before reaching a target state.

The index i in Definition 3 is allowed to be equal to 1 – the index of the first state – because if the agent starts in a reload state then taking no actions at all is a valid way to (trivially) reach a reload state. However, after initially reaching a reload state, the agent needs to keep visiting reload states indefinitely. For this, it is required that the agent takes at least one step to reach another reload state. Hence, for each of the above definitions – the functions $ReachCons_T(\cdot)$ and $ReachCons_T(\cdot,\cdot)$ and the vector $MinReach_T$ – we define versions where the index i from Definition 3 is required to be strictly greater than one. We denote these alternative versions in the same way but add a "+" superscript: $ReachCons_T^+(\cdot)$, $ReachCons_T^+(\cdot,\cdot)$, $MinReach_T^+$.

3.2 Safely reaching reload states

In this subsection we present an algorithm for computing the vector $MinInitCons$, which is defined to be equal to $MinReach_R^+$. Note that R was substituted for T in the subscript. The following truncation operator is used in Algorithm 1. Its purpose is explained below.

$$[\![\boldsymbol{x}]\!]_{\mathcal{M}}(s) := \begin{cases} \boldsymbol{x}(s) & \text{if } s \notin R \\ 0 & \text{if } s \in R \end{cases} \tag{2}$$

Algorithm 1 Computing $MinInitCons_{\mathcal{M}}$, taken from [3]

Input: a CMDP $\mathcal{M} = (S, A, \Delta, C, R, cap)$
Output: the vector $MinInitCons_{\mathcal{M}}$

1: initialise $\boldsymbol{x} \in \overline{\mathbb{N}}^{|S|}$ to be ∞ in all components
2: **repeat**
3: $\quad \boldsymbol{x}_{\text{old}} \leftarrow \boldsymbol{x}$
4: $\quad$ **for all** $s \in S$ **do**
5: $\quad\quad c \leftarrow \min_{a \in A}\Big(C(s,a) + \max_{s' \in Succ(s,a)} [\![\boldsymbol{x}_{\text{old}}]\!]_{\mathcal{M}}(s')\Big)$
6: $\quad\quad \boldsymbol{x}(s) \leftarrow \min(c, \boldsymbol{x}(s))$
7: $\quad$ **end for**
8: **until** $\boldsymbol{x}_{\text{old}} = \boldsymbol{x}$
9: **return** $\boldsymbol{x}$

3.2.1 Explanation of Algorithm 1

We will explain three aspects of Algorithm 1: the initialisation, the structure of the repeat-loop, and the expression on line 5. Concerning the initialisation: at no point did we require that it is possible to reach a reload state from each state. For a state from which the agent cannot reach a reload state, there is no finite energy level that guarantees reaching a reload state. Hence all components of x are initialised to ∞ in line 1 of Algorithm 1.

Regarding the structure of the repeat-loop: in line 3 the current approximation of $MinInitCons_{\mathcal{M}}$ is stored in a variable x_{old} so it can be compared against the next

approximation in line 8. When there is no longer any change after an iteration of the loop, i.e. two subsequent approximations are the same, the loop is terminated.

Finally, the expression in line 5 will be explained in steps. We want to assign to the variable c the lowest amount of energy which guarantees reaching a reload state. From the problem statements it should be clear that the "free variable" is the strategy. In other words, we optimise the result by finding the best strategy. And since strategies choose actions, this boils down to optimising the result by choosing actions. Therefore we can determine for each action the lowest[4] energy level that guarantees reaching a reload state, and subsequently take the minimum over all actions. The produced strategy should then simply be constructed such that it selects the action corresponding to that minimum.

Determining for a given action and state the lowest energy level that guarantees reaching a reload state, is done by the following expression (which is part of the expression in line 5).

$$C(s,a) + \max_{s' \in Succ(s,a)} [\![x_{\text{old}}]\!]_{\mathcal{M}}(s')$$

For each successor state s', we consider how much of the resource is needed to reach a reload state from s', using our previous approximation x_{old}. However, if s' is itself a reload state, this cost becomes zero. This is exactly what the truncation operator is for.

Reaching a reload state must be guaranteed. So regardless of what s' is randomly determined to be, the energy level must be sufficient. Hence we take the maximum over all successor states. Finally, we add the cost of taking action a: $C(s,a)$.

3.2.2 Correctness of Algorithm 1

The essence of Algorithm 1 is repeatedly applying a functional to a starting value. More concretely, the functional $\mathcal{G}$ is applied to $\boldsymbol{\infty}$, where

$$\mathcal{G}(v)(s) := \min_{a \in A} \left(C(s,a) + \max_{s' \in Succ(s,a)} [\![v]\!]_{\mathcal{M}}(s') \right)$$

(copied verbatim from [3]) and $\boldsymbol{\infty} \in \overline{\mathbb{N}}^{|S|}$ is a vector with all components equal to ∞. The counterparts of these definitions are

$$\mathcal{F}(v)(s) := \begin{cases} \min_{a \in A} \left(C(s,a) + \max_{s' \in Succ(s,a)} v(s') \right) & \text{if } s \notin R \\ 0 & \text{if } s \in R \end{cases}$$

and x_T, which is defined to be a vector with $x_T(s) = 0$ if $s \in T \subseteq S$ and $x_T(s) = \infty$ otherwise. Instead of directly proving that applying $\mathcal{G}$ to $\boldsymbol{\infty}$ gives the correct result, the approach is to prove that repeatedly applying $\mathcal{F}$ to x_R in a modified CMDP $\widetilde{\mathcal{M}}$ (defined below) gives the correct result and that the latter result equals the result of applying $\mathcal{G}$ to $\boldsymbol{\infty}$ in the original CMDP $\mathcal{M}$.

Theorem 1 ([3, Theorem 2])**.** *Denote by n the length of the longest simple path in $\mathcal{M}$. Then iterating $\mathcal{F}$ on x_T yields a fixed point in at most n steps and this fixed point equals $MinReach_T$. (Proof omitted.)*

[4] More accurately: our currently best approximation.

Reload states are assigned the value zero by $\mathcal{F}$: it does not require any energy to go to a reload state if the agent is already in a reload state. However, we want the agent to travel between reload states, which requires that at least one step is taken. The way we enfore this is by creating a modified CMDP.

Definition 5. Suppose $\mathcal{M} = (S, A, \Delta, C, R, cap)$ is a CMDP with $S = \{s_0, \ldots, s_N\}$. Let $\widetilde{S} = \{\tilde{s}_0, \ldots, \tilde{s}_N\}$ be a set that contains for each $s_i \in S$ a duplicate state $\tilde{s}_i$. Furthermore, define $\widetilde{\Delta}$ and $\widetilde{C}$ such that for all $a \in A$ and for $i = 0, \ldots, N$,

$$\widetilde{\Delta}(s_i, a) := \Delta(s_i, a)$$
$$\widetilde{\Delta}(\tilde{s}_i, a) := \Delta(s_i, a)$$
$$\widetilde{C}(s_i, a) := C(s_i, a)$$
$$\widetilde{C}(\tilde{s}_i, a) := C(s_i, a)$$

We define the CMDP $\widetilde{\mathcal{M}} = (S \cup \widetilde{S}, A, \widetilde{\Delta}, \widetilde{C}, R, cap)$. ♢

In Definition 5 the set of states is duplicated, but the set of reload states is not changed. This means that for each j such that s_j is a reload state, $\tilde{s}_j$ is not a reload state, but everything else is the same for $\tilde{s}$: after taking an abritrary action $a \in A$, the cost and the resulting probability distribution are the same as if the agent had taken action a in s_j. Therefore starting in a state $\tilde{s}_k$ is equivalent to starting in s_k, except the initial state will not be regarded as a reload state by $\mathcal{F}$.

Lemma 2 ([3, Lemma 2])**.** *Let $\mathcal{M}$ be a CMDP and let $\widetilde{\mathcal{M}}$ be the corresponding CMDP from Definition 5. Then $MinReach^{+}_{\mathcal{M},T}(s) = MinReach_{\widetilde{\mathcal{M}},T}(\tilde{s})$ for each state s of $\mathcal{M}$, where $\tilde{s}$ corresponds to s as in Definition 5. (Proof omitted.)*

Lemma 3 ([3, Lemma 3])**.** *Let $\boldsymbol{\infty} \in \overline{\mathbb{N}}^{|S|}$ be a vector with all components equal to ∞. Consider iterating $\mathcal{G}$ on $\boldsymbol{\infty}$ in $\mathcal{M}$ and $\mathcal{F}$ on x_R in $\widetilde{\mathcal{M}}$. Then for each $i \geq 0$ and each $s \in R$ we have $\mathcal{G}^i(\boldsymbol{\infty})(s) = \mathcal{F}^i(x_R)(\tilde{s})$ and for every $s \in S \setminus R$ we have $\mathcal{G}^i(\boldsymbol{\infty})(s) = \mathcal{F}^i(x_R)(s)$. (Proof omitted.)*

Theorem 4 ([3, Theorem 3])**.** *Algorithm 1 correctly computes the vector MinInitCons. Moreover, the repeat-loop terminates after at most $|S|$ iterations.*

Proof. In the repeat-loop, the operator $\mathcal{G}$ is iteratively applied to a starting point. We will show that this leads to a fixed point and that this fixed point equals *MinInitCons*.

Since the length of the longest simple path in $\mathcal{M}$ is at most $|S|$, we know from Theorem 1 that iterating $\mathcal{F}$ on x_R leads to a fixed point after at most $|S|$ steps. Let

$$\boldsymbol{\infty} \in \overline{\mathbb{N}}^{|S|}$$

be the vector with every element equal to ∞. We will first show that iterating $\mathcal{G}$ on $\boldsymbol{\infty}$ leads to a fixed point in at most as many steps as it takes when iterating $\mathcal{F}$ on x_R. Let i and j be the number of iterations after which $\mathcal{F}$ and $\mathcal{G}$ reach a fixed point, respectively. Suppose $\mathcal{G}$ has not yet reached a fixed point after k steps. This means there is some $s \in S$ such that

$$\mathcal{G}^{k+1}(\boldsymbol{\infty})(s) \neq \mathcal{G}^k(\boldsymbol{\infty})(s).$$

Then by Lemma 3

$$\mathcal{F}^{k+1}(x_R)(\tilde{s}) = \mathcal{G}^{k+1}(\boldsymbol{\infty})(s) \neq \mathcal{G}^{k}(\boldsymbol{\infty})(s) = \mathcal{F}^{k}(x_R)(\tilde{s})$$

so $\mathcal{F}$ has not reached a fixed point after k steps either. It follows that $j \leq i$.

Finally, the fact that the output is correct can be seen as follows:

$$\begin{aligned}
\mathcal{G}^{i}(\boldsymbol{\infty})(s) &= \mathcal{F}^{i}(x_R)(\tilde{s}) && \text{(Lemma 3)} \\
&= \mathit{MinReach}_{\widetilde{\mathcal{M}},R}(\tilde{s}) && \text{(Theorem 1)} \\
&= \mathit{MinReach}^{+}_{\mathcal{M},R}(s) && \text{(Lemma 2)} \\
&= \mathit{MinInitCons}(s) && \text{(By definition.)}
\end{aligned}$$

□

3.3 Solving the Safety problem

Since the goal is for the agent to survive indefinitely, it must be ensured that after any reload state it reaches, it can reach another. Therefore it is necessary to find a subset $R' \subseteq R$ of reload states such that from any $r' \in R'$ another reload state in R' can be reached in at least one step, with consumption at most cap. If we were to relax this requirement it could happen that the agent goes to a certain reload state from where it would take more than cap of the resource to reach another reload state.

Algorithm 2 Computing the vector *Safe*, taken from [3]

Input: a CMDP $\mathcal{M}$
Output: the vector *Safe*

$\mathit{cap} \leftarrow \mathit{cap}(\mathcal{M})$
$\mathit{Rel} \leftarrow R$
$\mathit{ToRemove} \leftarrow \emptyset$
repeat
  $\mathit{Rel} \leftarrow \mathit{Rel} \setminus \mathit{ToRemove}$
  $\boldsymbol{mic} \leftarrow \mathit{MinInitCons}_{\mathcal{M}(\mathit{Rel})}$
  $\mathit{ToRemove} \leftarrow \{r \in \mathit{Rel} \mid \boldsymbol{mic}(r) > \mathit{cap}\}$
until $\mathit{ToRemove} = \emptyset$

for all $s \in S$ **do**
  if $\boldsymbol{mic}(s) > \mathit{cap}$ **then**
    $\boldsymbol{out}(s) \leftarrow \infty$
  else
    $\boldsymbol{out}(s) \leftarrow \boldsymbol{mic}(s)$
  end if
end for
for all $r \in \mathit{Rel}$ **do**
  $\boldsymbol{out}(r) \leftarrow 0$
end for
return $\boldsymbol{out}$

In Algorithm 2 we use the notation $\mathcal{M}(\mathit{Rel})$ to refer to the CMDP that is equal to $\mathcal{M}$ except its set of reload states has been replaced by Rel. Then $\mathit{MinInitCons}_{\mathcal{M}(\mathit{Rel})}$ simply means the vector $\mathit{MinInitCons}$ corresponding to the CMDP $\mathcal{M}(\mathit{Rel})$.

Figure 2:

3.3.1 Explanation of Algorithm 2

In Algorithm 2 the set Rel is initially a copy of R (line 2). In lines 4–8 we repeatedly remove from Rel the reload states from which no other reload state can be reached (again with ≥ 1 step and $\leq cap$ consumption). The reason this has to be done multiple times is illustrated by the following example.

Example 1. Consider the CMDP in Figure 2 where there is only one action ($A = \{a\}$) and where each state has exactly one outgoing transition, which has probability 1. Each transition has cost $1 < cap$.

It is impossible to reach a reload state from s_2 after taking at least one step, let alone without consuming more than cap of the resource. As a result, $\boldsymbol{mic}(s_2)$ will be ∞ in the first iteration of the repeat-until loop in Algorithm 2. On the other hand, we will have $\boldsymbol{mic}(s_1) = 1$ because of the transition to s_2. So after one iteration of the repeat-until loop, s_2 will have been removed from Rel, but not s_1.

However, if the agent is in s_1 it will certainly go to s_2 in its next step. And from s_2 it will not reach a reload state. So $s_1 \notin R'$ and s_1 should be removed from Rel. It was not removed in the first iteration, since it could reach a reload state (s_2), but after s_2 is removed and $\boldsymbol{mic}(s_1)$ becomes ∞, another iteration is necessary to remove s_1 as well. ♢

After line 8, Rel satisfies the requirement for R' described above, since

$$\{r \in Rel \mid \boldsymbol{mic}(r) > cap\} = \emptyset. \tag{3}$$

This means that $\boldsymbol{mic} = MinInitCons_{\mathcal{M}(R')}$ after line 8. Then in lines 9–15 any value in $\boldsymbol{mic}$ that exceeds the capacity is replaced by ∞ as is required by the definition of the vector $Safe$. Finally, in lines 16–18 the output values are set to zero for reload states from Rel. The reason is that if the agent starts in one of those states, then by (3) it is possible to guarantee reaching another reload state from Rel with at least one step and consuming at most cap. However, the states in Rel are reload states and hence by Remark 1 it is enough to start with zero resource. The pseudocode in lines 16–18 was omitted in [3], but included in their implementation.

3.3.2 Correctness

Theorem 5 ([3, Theorem 4])**.** *Algorithm 2 computes the vector Safe in polynomial time.*

Proof. In each iteration of the repeat-loop at least one state is removed from Rel. Since Rel is initialised to R and the number of states is finite, the repeat-loop takes at most $|R| < \infty$ iterations. Furthermore, by Theorem 4 computing $MinInitCons$ in line 6 takes at most $|S|$ steps. It follows that the time complexity is polynomial.

For the correctness, we first prove the component-wise inequality

$$\boldsymbol{out} \leq \mathit{Safe}. \tag{4}$$

Suppose $s \in S$. If $s \in \mathit{Rel}$ then $\boldsymbol{out}(s) = 0$ and we are done, so assume $s \notin \mathit{Rel}$. If $\mathit{Safe}(s) = \infty$ then (4) holds for s, so assume $\mathit{Safe}(s)$ is finite. By definition this means

$$\mathit{Safe}(s) \leq \mathit{cap}.$$

If we can prove that

$$\boldsymbol{mic}(s) \leq \mathit{Safe}(s) \tag{5}$$

then by lines 10–14, $\boldsymbol{out}(s) = \boldsymbol{mic}(s)$ and we are done. So it is sufficient to prove that (5) holds when the algorithm terminates.

Above, we have reduced Inequality (4) to (5). This will now be reduced further to (6) below. For a CMDP $\mathcal{M}'$, $\mathit{Safe}_{\mathcal{M}'}(s)$ is the minimal amount of initial fuel to move around indefinitely without running out of energy and $\mathit{MinInitCons}_{\mathcal{M}'}(s)$ is the minimal amount of initial fuel to reach a reload state in at least one step. So if the agent has enough energy to do the former, it also has enough for the latter. From this it follows that

$$\mathit{MinInitCons}_{\mathcal{M}'}(s) \leq \mathit{Safe}_{\mathcal{M}'}(s).$$

Now since $\boldsymbol{mic} = \mathit{MinInitCons}_{\mathcal{M}(\mathit{Rel})} \leq \mathit{Safe}_{\mathcal{M}(\mathit{Rel})}$ it suffices to prove that

$$\mathit{Safe}_{\mathcal{M}(\mathit{Rel})} \leq \mathit{Safe}_{\mathcal{M}} \tag{6}$$

is an invariant of the algorithm.

Finally, we reduce (6) to proving that at every point of execution

$$\mathit{Safe}_{\mathcal{M}}(t) = \infty \text{ for all } t \in R \setminus \mathit{Rel}. \tag{7}$$

Equation (7) is proved in the next paragraph. But first, to see that (7) is sufficient, suppose it holds and let $s' \in S$. The only way that the values of $\mathit{Safe}_{\mathcal{M}(\mathit{Rel})}$ could be affected by a state $t \in R \setminus \mathit{Rel}$ being a reload state, is if there is a way to go to t and subsequently survive indefinitely by moving between reload states. If it is cheap to go to t, then this could lower the value of $\mathit{Safe}_{\mathcal{M}(\mathit{Rel})}(s')$. However, if the agent reaches t, it cannot be guaranteed that it will reach another reload state using at most cap energy. In other words, there does not exist a strategy that is d-safe (with $d \leq \mathit{cap}$) that would choose an action a at some state $\hat{s}$ such that $t \in \mathit{Succ}(\hat{s}, a)$. Otherwise, the strategy would not be d-safe for any d. In conclusion, it does not matter whether t is a reload state and declaring all states in $R \setminus \mathit{Rel}$ to be non-reload, does not change the values of $\mathit{Safe}_{\mathcal{M}(\mathit{Rel})}$ compared to $\mathit{Safe}_{\mathcal{M}}$. Hence $\mathit{Safe}_{\mathcal{M}}(s') = \mathit{Safe}_{\mathcal{M}(\mathit{Rel})}(s')$.

Now we know that (7) implies (6), but we still need to prove (7). We will use induction on the set Rel. Denote by Rel_i the set Rel after the i-th iteration. For $i = 0$, $\mathit{Rel}_i = R$ so the statement trivially holds. Suppose $i \geq 1$ and let $s \in R \setminus \mathit{Rel}_i$. If we can show that for no strategy σ there exists a $d \leq \mathit{cap}$ such that σ is d-safe in s, then by definition $\mathit{Safe}_{\mathcal{M}}(s) = \infty$. So assume σ is a strategy. Suppose a run in $\mathit{Comp}(\sigma, s)$ were to visit a state t in $R \setminus \mathit{Rel}_{i-1}$. Then it is possible to reach t when playing by σ and it cannot be guaranteed that the resource level stays non-negative after t, since $\mathit{Safe}_{\mathcal{M}}(t) = \infty$ by the induction hypothesis. Therefore σ is not cap-safe. Now suppose all runs in $\mathit{Comp}(\sigma, s)$ only visit reload states in Rel_{i-1}. This means that

$s \in Rel_{i-1} \setminus Rel_i = ToRemove$ because of the assumption on s. From lines 6–7 we know that $MinInitCons_{\mathcal{M}(Rel_{i-1})}(s) > cap$ which implies that there is a run $\rho \in Comp(\sigma, s)$ such that $ReachCons^{+}_{Rel_{i-1}}(\rho) > cap$. It follows that ρ is not cap-safe. The details for this last step can be found in the proof of [3, Theorem 4]. As a result, σ is not cap-safe, since there is a possible run that is not cap-safe.

Finally, for the correctness it remains to show that

$$\boldsymbol{out} \geq Safe \tag{8}$$

after the algorithm terminates. Assume $s \in S$. If $\boldsymbol{out}(s) = \infty$ then (8) follows immediately, so assume $\boldsymbol{out}(s) < \infty$. Then by lines 10–14 we have $\boldsymbol{out}(s) \leq cap$. If the agent starts in s with $\boldsymbol{out}(s)$ units of resource, it has enough to reach a reload state from Rel.[5] By construction of Rel, the agent then has enough resource to guarantee reaching another reload state from Rel while consuming at most cap and can keep visiting reload states from Rel like that indefinitely. It follows that $\boldsymbol{out}(s) \leq cap$ is enough initial resource for resource exhaustion to be prevented, meaning $\boldsymbol{out}(s) \geq Safe(s)$. □

Definition 6. (Partly copied almost verbatim from [3].) An action a is *safe in a state s* with $Safe(s) < \infty$ if

$$s \notin R \quad \text{and} \quad C(s,a) + \max_{t \in Succ(s,a)} Safe(t) \leq Safe(s) \tag{9}$$

$$\text{or} \quad s \in R \quad \text{and} \quad C(s,a) + \max_{t \in Succ(s,a)} Safe(t) \leq cap. \tag{10}$$

This means that if the agent has $Safe(s)$ (or cap) units of energy, it has enough energy to take action a and subsequently reach a reload state. The reason to have a special case for $s \in R$ is that it might be possible to guarantee survival with $d < cap$ energy and then we would have $Safe(s) < cap$. However, if $s \in R$, cap units of energy are available and we can also consider actions to be safe if they require d' units of energy for survival, where $Safe(s) < d' \leq cap$.

Note that by definition of $Safe$, there is at least one safe action for states with $Safe(s) < \infty$. For a state s with $Safe(s) = \infty$, all actions are said to be safe in s. ♢

In Theorem 6, which shows we have solved the second part of the Safety problem, the use of the word "memoryless" might seem inaccurate since the strategy needs to remember a safe action for each state. However "memoryless" refers to the fact that the strategy does not need to know the whole history: it only needs to know the last state, i.e. the current state.

Theorem 6 ([3, Theorem 5])**.** *Any strategy which always selects an action that is safe in the current state is $Safe_{\mathcal{M}}(s)$-safe in every state s. In particular, in each CMDP $\mathcal{M}$ there is a memoryless strategy σ that is $Safe_{\mathcal{M}}(s)$-safe in every state s. Moreover, σ can be computed in polynomial time.*

Proof. Suppose σ is a strategy that always selects a safe action in the current state. Suppose $s \in S$ is a state with $Safe_{\mathcal{M}}(s) < \infty$. Let $d := Safe_{\mathcal{M}}(s) \leq cap$ and suppose $\rho \in Comp(\sigma, s)$ is a run that is not d-safe. Then there is an index i such that $RL_d(\rho_{\ldots i}) = \bot$.

[5] This is because $\boldsymbol{out}(s) < \infty$ implies either (i) we have $\boldsymbol{out}(s) = \boldsymbol{mic}(s)$ by lines 10–14, meaning the agent has enough of the resource to reach an $r \in Rel$ in at least one step, or (ii) we have $\boldsymbol{out}(s) = 0$, meaning $s \in Rel$ by lines 16–18 so the agent is already in Rel.

No strategy can be $\bot$-safe, since the resource level in a run would always be $\bot$ if it were initialised with $\bot$. Hence, $Safe_{\mathcal{M}}(t) > \bot$ for all $t \in S$ and so there is an index $j \leq i$ such that $RL_d(\rho_{...j}) < Safe_{\mathcal{M}}(\rho_j)$. That is, eventually the current resource level drops below the $Safe_{\mathcal{M}}$ value of the current state.

We will now derive a contradiction by proving, using induction, that $RL_d(\rho_{...k}) \geq Safe_{\mathcal{M}}(\rho_k)$ for all $k \geq 1$. In the case $k = 1$ we have equality. Suppose $k > 1$. We need to say something about $RL_d(\rho_{...k})$ so let us simplify the matter by eliminating the first case in (1). By the induction hypothesis, $RL_d(\rho_{...k-1}) \geq Safe_{\mathcal{M}}(\rho_{k-1}) > \bot$. Let $a := \sigma(\rho_{...k-1})$ be the last action that σ selected before the agent reached state ρ_k. By assumption, a is safe in ρ_{k-1}. Therefore if $\rho_{k-1} \notin R$ we have by (9) and the induction hypothesis that,

$$\begin{aligned} C(\rho_{k-1}, a) &\leq C(\rho_{k-1}, a) + Safe(\rho_k) \\ &\leq C(\rho_{k-1}, a) + \max_{t \in Succ(\rho_{k-1}, a)} Safe(t) \\ &\leq Safe(\rho_{k-1}) \\ &\leq RL_d(\rho_{...k-1}). \end{aligned}$$

If $\rho_{k-1} \in R$ then

$$\begin{aligned} C(\rho_{k-1}, a) &\leq C(\rho_{k-1}, a) + Safe(\rho_k) \\ &\leq C(\rho_{k-1}, a) + \max_{t \in Succ(\rho_{k-1}, a)} Safe(t) \\ &\leq cap \\ &= RL_d(\rho_{...k-1}). \end{aligned}$$

by (10) and Definition 2. In conclusion, $RL_d(\rho_{...k-1}) \neq \bot$ and $C(\rho_{k-1}, a) \leq RL_d(\rho_{...k-1})$ so the first case in (1) is eliminated.

Now we will make the induction step. Suppose $\rho_k \in R$. By (10) we have

$$\begin{aligned} RL_d(\rho_{...k}) &= cap \\ &\geq C(\rho_{k-1}, a) + \max_{t \in Succ(\rho_{k-1}, a)} Safe(t) \\ &\geq Safe(\rho_k). \end{aligned}$$

If $\rho_k \notin R$, then by (1), the induction hypothesis, and (9),

$$\begin{aligned} RL_d(\rho_{...k}) &= RL_d(\rho_{...k-1}) - C(\rho_{k-1}, a) \\ &\geq Safe(\rho_{k-1}) - C(\rho_{k-1}, a) \\ &\geq \max_{t \in Succ(\rho_{k-1}, a)} Safe(t) \\ &\geq Safe(\rho_k). \end{aligned}$$

This proves that $RL_d(\rho_{...k}) \geq Safe_{\mathcal{M}}(\rho_k) > \bot$ for all $k \geq 1$, which gives a contradiction with $RL_d(\rho_{...i}) = \bot$. It follows that all runs in $Comp(\sigma, s)$ are $Safe_{\mathcal{M}}(s)$-safe and therefore σ is $Safe_{\mathcal{M}}(s)$-safe.

Next, we prove that in each CMDP $\mathcal{M}$ there exists a strategy that is $Safe_{\mathcal{M}}$-safe in each state s. Let $f : S \to A$ be a function that maps each state to a safe action. Define the strategy σ' by $\sigma'(h) := f(last(h))$. Then $\sigma'(h) = \sigma'(last(h))$, so σ' is memoryless.

Furthermore, σ' always selects an action that is safe in the current state so by the above argument, σ' is $Safe_{\mathcal{M}}(s)$-safe in every state s.

Finally, we prove the time complexity. To compute σ' we first execute Algorithm 2 which by Theorem 5 has polynomial complexity. Then for each state-action pair we check one of (9) and (10). The number of state-action pairs is polynomial and the maxima in Definition 6 can be computed in polynomial time. It follows that the overall time complexity is polynomial. □

4 Counter selectors

In this section, we define *counter selectors*, which can be used to represent strategies with a finite amount of memory. We say that a strategy σ is a *finite-memory strategy* if σ can be represented by a *memory structure*, which is a tuple (M, nxt, up, m_0) where

- M is a finite set, the elements of which we call *memory elements*;
- $nxt\colon M \times S \to A$ is a function that chooses the next action to be taken by the agent;
- $up\colon M \times S \times A \times S \to M$ is a function that updates the "current" memory element;[6] and
- $m_0\colon S \to M$ is a function which selects an initial memory element based on the initial state.

It is convenient to define a function $up^*\colon M \times Hist \to M$ which is semantically similar to up, except it takes a history as its second argument instead of a state, action, and a state as its second, third, and fourth argument, respectively. The following definition was copied verbatim from [3].

$$up^*(m, \alpha) := \begin{cases} m & \text{if } \alpha = s \text{ has length 0} \\ up(up^*(m, \beta), last(\beta), a, t) & \text{if } \alpha = \beta a t \text{ for some } a \in A \text{ and } t \in S \end{cases}$$

up is a function which returns the new (current) memory element based on the old (previous) memory element, the current state, the last action taken, and the state which the agent happened to visit next. On the other hand, up^* is a function which returns the current memory element based on the initial memory element and the agent's history. Applying up repeatedly will give exactly the same result as applying up^* once to the whole history.

If $\mu = (M, nxt, up, m_0)$ is a memory structure, we can define a strategy σ_μ by $\sigma_\mu(\alpha) = nxt(up^*(m_0(s_1), \alpha), s_n)$ for each history $\alpha = s_1 a_1 s_2 a_2 \dots s_n$. Here we apply m_0 to the initial state to get the initial memory element. Subsequently, we apply up^* to the initial memory element and the input history to get the current memory element. Finally, we apply nxt to the current memory element and the last (i.e. current) state s_n to determine which action to take next. We say that σ_μ is *encoded by* μ.

To prevent resource exhaustion, strategies need to track the energy level of the agent. This seems to contrast Theorem 6, which mentions memoryless strategies. However, to track the energy level, strategies do not need to remember the whole

[6] The memory structure does not actually contain a memory element, but the "current" memory element can be found for each finite prefix of a path by repeatedly applying up.

history, which is what "memoryless" refers to. Instead, strategies can remember the the energy level using an integer counter. This integer is updated when the agent takes an action and afterwards the strategy can "forget" that part of the history.

The aforementioned integer counter is of course bounded by the capacity and hence it requires only finite memory. Strategies with such counters are called *finite-counter strategies*. Finite-counter strategies choose actions to play using *selection rules*.

Definition 7. A *selection rule* for $\mathcal{M}$ is a function $\phi\colon \{0, 1, \dots, cap(\mathcal{M})\} \to A \cup \{\bot\}$ where the value $\bot$ is meant to indicate the result being "undefined." ♢

The set of resource levels for which ϕ is not $\bot$ ("undefined") is denoted by $dom(\phi) := \{x \in \{0, \dots, cap\} \mid \phi(x) \neq \bot\}$ which is different from what the domain of ϕ would be according to the definition of domain common in mathematics. $Rules_{\mathcal{M}}$ is the set of all selection rules for $\mathcal{M}$. We omit the subscript if $\mathcal{M}$ is clear from context.

When selecting actions according to a selection rule ϕ, at any point the next action to play is

$$\phi(\max\{x \in dom(\phi) \mid x \leq c\}), \tag{11}$$

where c is the current energy level. In words, we first disregard all possible energy levels for which ϕ is "undefined" ($\bot$). Then, we take the maximum energy level x which is no greater than the current energy level. Finally, we apply ϕ to x and obtain the next action.

Definition 8. (Copied verbatim from [3].)
A *counter selector* for $\mathcal{M}$ is a function $\Sigma\colon S \to Rules$. ♢

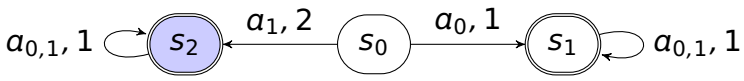
We want to create a strategy from a counter selector, but we need to know the current energy level in order to choose an action using a selection rule. Therefore we need to use a memory structure to keep track of the resource level. Let Σ be a counter selector. Let $r \in \{0, \dots, cap(\mathcal{M})\}^{|S|}$ be a vector which for each state holds an initial energy level. To ensure that the strategy we create is unique, we fix an action $a_0 \in A$ which we will use when a certain variable (n) below does not exist. Finally, we assume $\bot < k$ for all $k \in \mathbb{N}$. Now we denote by Σ^r the strategy encoded by the memory structure (M, nxt, up, m_0) which we define as follows.

- $M := \{\bot\} \cup \{0, \dots, cap(\mathcal{M})\}$ is the set of all possible resource levels, including $\bot$ which represents an insufficient level.
- For $m \in M$, $s \in \mathbb{N}$, let $n \in dom(\Sigma(s))$ be the greatest resource level such that $n \leq m$. If n exists, we let $nxt(m, s) := n$ and otherwise $nxt(m, s) := a_0$. In other words, we apply (11) with $c = m$ but if the set $\{x \in dom(\phi) \mid x \leq m\}$ is empty, we let the result be a_0.
- Suppose $m \in M$; $s, t \in S$; $a \in A$. The following definition was copied verbatim from [3] and is very similar to Equation (1) from Definition 2.

$$up(m, s, a, t) := \begin{cases} m - C(s, a) & \text{if } s \notin R \text{ and } C(s, a) \leq m \neq \bot \\ cap(\mathcal{M}) - C(s, a) & \text{if } s \in R \text{ and } C(s, a) \leq cap(\mathcal{M}) \text{ and } m \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Here the current resource level, of which we keep track using m, stays insufficient ($\bot$) if it already was and becomes insufficient if the cost of playing action a

Figure 3: an example CMDP where $SafePR_T(s_0) > Safe(s_0)$. s_1 and s_2 are reload states, but only s_2 is a target state. The transition labels indicate actions and cost respectively.



is greater than the amount of resource the agent had in s. If there is no reason to say that the resource level is insufficient, we simply subtract the cost of playing a.

- We define m_0 by $m_0(s) := r(s)$.

If σ is a strategy and there exist a counter selector Σ and a vector r' (from the same set as above), such that $\sigma = \Sigma^{r'}$, then σ is a finite-counter strategy since the energy level is kept track of by the memory structure using a memory element.

The above construction will be used to create a strategy that solves the positive reachability problem. The vector $SafePR_T$ will be substituted for r. As can be seen above, the memory element is initialised to $r(s)$, which in this case would be $SafePR_T(s)$. This means that the strategy assumes the initial resource level in s is $SafePR_T(s)$, which might seem strange since this is not necessarily true. However, the counter selector, and therefore the strategy, will be constructed so that it is T-positive $SafePR_T(s)$-safe in s. Hence it can do its job if given $SafePR_T(s)$ units of resource. If the agent has more resource at the start, the strategy will assume the initial level is $SafePR_T(s)$ regardless. The extra resource will not get in the way. If the agent has less, it does not matter what the strategy does since it is not required to prevent resource exhaustion or give the agent a non-zero chance of reaching a target state if the initial resource level is insufficient.

5 Positive reachability

After computing the vector $Safe$, the next step is to determine $SafePR_T$. Loosely, for a state s the required initial energy to survive is $Safe(s)$ while the required initial energy to survive *and* reach a target state with non-zero probability is $SafePR_T(s)$. The reason $SafePR_T$ is not necessarily equal to $Safe$ and therefore needs to be computed separately, is illustrated by the example CMDP in Figure 3.

In this section we present and prove correct an algorithm to compute the vector $SafePR_T$ and, omitting some details which have already been explained, a strategy for which at least one of the resulting runs visits a set $T \subseteq S$ of target states. We fix a CMDP $\mathcal{M}$ for the rest of this section.

We introduce the following function, with "SPR" being short for "safe positive reachability."

$$SPR\text{-}Val_{\mathcal{M}} : S \times A \times \overline{\mathbb{N}}^{|S|} \to \overline{\mathbb{N}},$$
$$(s, a, \boldsymbol{x}) \mapsto C(s, a) + \min_{t \in Succ(s,a)} \big(\max(sos)\big)$$
$$\text{where } sos := \{\boldsymbol{x}(t)\} \cup \{Safe_{\mathcal{M}}(t') \mid t' \in Succ(s, a), t' \neq t\}$$

Here $\boldsymbol{x}$ should be thought of as a vector of resource levels. The set sos contains the resource level $\boldsymbol{x}(t)$ corresponding to t as well as the value $Safe_{\mathcal{M}}(t')$ for each potential successor t' other than t. This means that if the agent is in s and has $C(s,a)+\max(sos)$ units of the resource, it has enough to take action a and to either end up at t with $\boldsymbol{x}(t)$ amount of resource left, or to survive after reaching a different state t'. Hence the name sos: "succeed or survive." To be mathematically correct we would have to attach $t, s, a,$ and $\boldsymbol{x}$ as subscripts to sos, but this was omitted in favour of visual clarity.

We additionally take the minimum over all potential successors $t \in Succ(s,a)$ so that $SPR\text{-}Val(s,a,\boldsymbol{x})$ is the minimum amount of resource needed to guarantee that, for a certain successor $\hat{t}$, the agent either reaches $\hat{t}$ with $\boldsymbol{x}(\hat{t})$ units of energy remaining in its reserve, or it survives after ending up at a different successor $t' \neq \hat{t}$.

We define a two-sided version of the truncation operator defined in (2). This definition was copied verbatim from [3].

$$[\![\boldsymbol{x}]\!]_{\mathcal{M}}(s) := \begin{cases} \infty & \text{if } \boldsymbol{x}(s) > cap(\mathcal{M}) \\ \boldsymbol{x}(s) & \text{if } \boldsymbol{x}(s) \leq cap(\mathcal{M}) \text{ and } s \notin R \\ 0 & \text{if } \boldsymbol{x}(s) \leq cap(\mathcal{M}) \text{ and } s \in R \end{cases}$$

Algorithm 3 Positive reachability of T in $\mathcal{M}$, from [3]

Input: a CMDP $\mathcal{M}$ and a subset $T \subseteq S$ of target states
Output: the vector $SafePR_T$ and a corresponding counter selector Σ

$\boldsymbol{r} \leftarrow \{\infty\}^{|S|}$
for all $s \in S$ with $Safe_{\mathcal{M}}(s) < \infty$ **do**
  $\Sigma(s)(Safe_{\mathcal{M}}(s)) \leftarrow$ arbitrary action safe in s
end for
for all $t \in T$ **do**
  $\boldsymbol{r}(t) \leftarrow Safe_{\mathcal{M}}(t)$
end for

repeat
  $\boldsymbol{r}_{\text{old}} \leftarrow \boldsymbol{r}$
  for all $s \in S \setminus T$ **do**
    $\boldsymbol{a}(s) \leftarrow \arg\min_{a \in A} SPR\text{-}Val(s,a,\boldsymbol{r}_{\text{old}})$
    $\boldsymbol{r}(s) \leftarrow \min_{a \in A} SPR\text{-}Val(s,a,\boldsymbol{r}_{\text{old}})$
  end for
  $\boldsymbol{r} \leftarrow [\![\boldsymbol{r}]\!]_{\mathcal{M}}$
  for all $s \in S \setminus T$ **do**
    if $\boldsymbol{r}(s) < \boldsymbol{r}_{\text{old}}(s)$ **then**
      $\Sigma(s)(\boldsymbol{r}(s)) \leftarrow \boldsymbol{a}(s)$
    end if
  end for
until $\boldsymbol{r}_{\text{old}} = \boldsymbol{r}$
return $\boldsymbol{r}, \Sigma$

5.1 Explanation of Algorithm 3

5.1.1 Initialisation

In Algorithm 3, the vector $\boldsymbol{r}$ is to become $SafePR_T =: \boldsymbol{v}$ and we want to construct a counter selector Σ such that we can create a strategy $\Sigma^{\boldsymbol{v}}$ from it that is T-positive $\boldsymbol{v}(s)$-safe in each state s. Just like in Algorithm 1, we initialise the output vector, in this case $\boldsymbol{r}$, to ∞ in every component so that the result is correct for states where we do not find a way to reach the objective with a finite amount of initial resource.

The goal with positive reachability is to survive indefinitely and to reach a target state with non-zero probability. Hence, if the agent starts in a target state $t \in T$, the latter objective is immediately satisfied and the only thing left to be concerned with, is survival. This is why in line 6 the value of $\boldsymbol{r}$ is set to that of $Safe_{\mathcal{M}}$ for all target states. That is also why in the remainder of Algorithm 3 the value of $\boldsymbol{r}$ is only changed for non-target states. See lines 10 and 15.

For a counter selector Σ and a state s, $\Sigma(s)$ is the selection rule assigned to s. Using pseudocode notation, $\Sigma(s)(d) \leftarrow a$ essentially[7] means that if the agent is in s and its current resource level is at least d, it must take action a. In lines 2–4 we tell the agent that if its resource level is enough to prevent resource exhaustion from its current state, it should take some safe action to indeed prevent resource exhaustion.

5.1.2 Improving the intermediate approximations

As in Algorithm 1 there is a repeat-loop where the values of the result vector – and in this case the counter selector – are updated repeatedly until there are no more changes. In line 12 we try to find a better approximation of $SafePR_T$ for each non-target state. There we have the subexpression $SPR\text{-}Val(s, a, \boldsymbol{r}_{\text{old}})$. Consider the same subexpression but with $\boldsymbol{r}_{\text{old}}$ replaced by $SafePR_T$:

$$SPR\text{-}Val(s, a, SafePR_T). \tag{12}$$

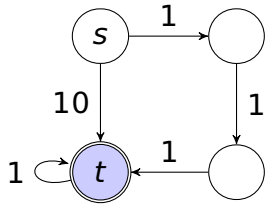
This is the minimal amount of resource for the agent to either not run out of energy or to reach a successor s' with at least $SafePR_T(s')$ resource remaining. However, having $SafePR_T(s')$ units of resource remaining in s' also means it can be guaranteed that the agent will not run out of energy, since there exists a strategy σ that is T-positive $SafePR_T(s')$-safe in s'. This means that with $SPR\text{-}Val(s, a, SafePR_T)$ units of resource, resource exhaustion can be prevented from s. Furthermore, there is a non-zero chance that the agent will visit s' and if the agent plays by σ, the chance that the agent will eventually visit a target state $t \in T$ from s' is not zero. Therefore (12) is close to $SafePR_T(s)$, except the latter is the *minimal* amount to guarantee survival with a non-zero chance of reaching a target state. To get the minimal amount we take the minimum over all actions.

The above example, where we replaced $\boldsymbol{r}_{\text{old}}$ by $SafePR_T$ to get (12), should give an idea of why we improve our approximations $\boldsymbol{r}$ of $SafePR_T$ by taking minima over $SPR\text{-}Val$ values. But we did not explain why the approximations get closer to $SafePR_T$. Before the repeat-loop, the value of $\boldsymbol{r}$ is equal to $SafePR_T$ for all target states[8] and ∞ for all other states. So $\boldsymbol{r} \geq SafePR_T$ and the approximations $\boldsymbol{r}$ need to decrease to get

[7] Unless there is some $e \in dom(\Sigma(s))$ with $d < e \leq c$, where c is the agent's current resource level.

[8] See Subsection 5.1.1.

Figure 4:



closer to $SafePR_T$. In line 12 we take the minimum over actions, so $SPR\text{-}Val(s, a, \boldsymbol{r}_{\text{old}})$ needs to decrease for some action a before $\boldsymbol{r}(s)$ can decrease.

This could happen as follows. Suppose $s \in S$ and the algorithm is currently in iteration $i > 1$ of the repeat-loop. If the value $\boldsymbol{r}(\hat{t})$ for some successor $\hat{t} \in Succ(s, a)$ was lowered in the previous iteration $i-1$, then $\min_{t \in Succ(s,a)}\big(\max(sos)\big)$ – from the definition of $SPR\text{-}Val$ – could decrease. In that case $SPR\text{-}Val(s, a, \boldsymbol{r}_{\text{old}})$ and hence $\boldsymbol{r}(s)$ would decrease. Then in the next iteration $i+1$, the value $\boldsymbol{r}(s')$ could decrease for some state s' which has s as a potential successor. This process continues.

The first time $\boldsymbol{r}(s)$ is decreased some state s, $\boldsymbol{r}(s)$ is not yet necessarily equal to $SafePR_T(s)$. For example, if there is an action a such that starting in s, survival and reaching a target state t with $\boldsymbol{r}(t) \leq cap$ can be guaranteed, where t is a direct successor of s, then in the first iteration of the repeat-loop $\boldsymbol{r}(s)$ may go from ∞ to a finite value at most cap. However, it could be that $C(s, a)$ is high and that there are a series of cheap transitions which also lead to t. See Figure 4. In that case $\boldsymbol{r}(s)$ might be lowered again after a few more iterations of repeat-loop.

Another way that for some state s the value of $\boldsymbol{r}$ can be lowered is if $\boldsymbol{r}(s) \leq cap$ and s is a reload state. In that case, in s, some amount $d \leq cap$ of resource is sufficient to guarantee survival with a non-zero chance of reaching T. But when leaving s the agent will have cap resource, even if it had zero when entering s. So we can set $\boldsymbol{r}(s)$ to zero using the two-sided truncation operator $[\![\cdot]\!]$. On the other hand, when we find a way to guarantee survival with a non-zero chance of reaching T for a state s with an amount of resource that is finite yet greater than cap, we set $\boldsymbol{r}(s)$ to ∞ using $[\![\cdot]\!]$ because this is required by the definition of $SafePR_T$.

Concerning the counter selector: the action corresponding to the minimum found for s in line 12 is stored in $\boldsymbol{\alpha}(s)$ and if the value $\boldsymbol{r}(s)$ was actually improved in the current iteration, line 17 is executed. As explained in Subsection 5.1.1, the effect of assigning to the counter selector as in line 17 is that if the agent is in s and its resource level is at least (the current value of) $\boldsymbol{r}(s)$ but strictly less than the first higher element of $dom(\Sigma(s))$, then it takes action $\boldsymbol{\alpha}(s)$. In other words, we have found a cheaper way to guarantee survival and a non-zero chance of reaching T, and we tell the agent to take that way if it has enough resource for it.

5.2 Correctness of Algorithm 3

The following functionals correspond to certain instructions in Algorithm 3. It is convenient to define them so we can refer to them in statements below. These two defin-

itions were copied verbatim from [3].

$$\mathcal{A}_{\mathcal{M}}(\boldsymbol{r})(s) := \begin{cases} Safe_{\mathcal{M}}(s) & \text{if } s \in T \\ \min_{a \in A}(SPR\text{-}Val_{\mathcal{M}}(s,a,\boldsymbol{r})) & \text{otherwise} \end{cases}$$

$$\mathcal{B}_{\mathcal{M}}(\boldsymbol{r}) := [\![\mathcal{A}_{\mathcal{M}}(\boldsymbol{r})]\!]_{\mathcal{M}}$$

Suppose $s \in S$ is a state. Then $SafePR_{\mathcal{M},T}(s)$ is the minimum amount of energy d such that there exists a strategy σ which is d-safe in s and which produces at least one run ρ that starts in s and visits T *at some point*. That is, there is no restriction on when the agent visits T. We now define the vector $SafePR^{i}_{\mathcal{M},T}$ which is the same as $SafePR_{\mathcal{M},T}$, except the run ρ from above is required to visit T in the first i steps.

The last definition we need for Lemma 7 is the vector $\boldsymbol{y}_T$, which is equal to $Safe_{\mathcal{M}}$ except the values of non-target states become ∞. (Definition copied verbatim from [3].)

$$\boldsymbol{y}_T(s) := \begin{cases} Safe_{\mathcal{M}}(s) & \text{if } s \in T \\ \infty & \text{if } s \notin T \end{cases}$$

Iterating the operator $\mathcal{B}$ on the vector $\boldsymbol{y}_T$ corresponds to iterating the repeat-loop of Algorithm 3 on what the value of $\boldsymbol{r}$ is just before the repeat-loop begins. Lemma 7 says that after the i-th iteration of operator $\mathcal{B}$ on the vector $\boldsymbol{y}_T$, the result is the vector $SafePR^{i}_{\mathcal{M},T}$. Hence, the latter is also the value of $\boldsymbol{r}$ after the i-th iteration of the repeat-loop.

Lemma 7 ([3, Lemma 4])**.** *Consider the iteration of $\mathcal{B}_{\mathcal{M}}$ on the initial vector $\boldsymbol{y}_T$. Then for each $i \geq 0$ it holds that $\mathcal{B}^{i}_{\mathcal{M}}(\boldsymbol{y}_T) = SafePR^{i}_{\mathcal{M},T}$. (Proof omitted.)*

Theorem 8 ([3, Theorem 6])**.** *Algorithm 3 always terminates after a polynomial number of steps and upon termination, $\boldsymbol{r} = SafePR_T$.*

Proof. $\boldsymbol{r}$ is set to $\boldsymbol{y}_T$ in the part of Algorithm 3 before the repeat-loop: lines 1–7. Furthermore, executing the repeat-loop is equivalent to iterating operator $\mathcal{B}$. In [3, Lemma 5] it is proved that $\mathcal{B}^{K}_{\mathcal{M}}(\boldsymbol{y}_T) = SafePR_T$ where $K = |R| + (|R|+1)\cdot(|S|-|R|+1)$. It follows that after K iterations, $\boldsymbol{r} = SafePR_T$. The time complexity follows from the fact that K is of polynomial size. □

It is more difficult to prove that from the produced counter selector a strategy can be generated that is T-positive $SafePR_T(s)$-safe in each state s. We will not give all the details. The proof is done using invariants. First, it can be shown that at the end of each iteration of the repeat-loop, $\boldsymbol{r} \geq Safe$ [3, Lemma 6]. Next, [3, Lemma 7] shows that a strategy $\Sigma^{\boldsymbol{y}}$ generated from a counter selector Σ is $Safe(s)$-safe in each state s if $\boldsymbol{y} \geq Safe$ and Σ, put simply, always keeps the agent's current resource level at least as high as the value of $Safe$ corresponding to the agent's current state. By combining the former invariant with the latter statement, it follows that after each iteration of the repeat-loop, the strategy $\Sigma^{\boldsymbol{r}}$ is $Safe(s)$-safe in each state s.

The above explains how it can be shown that the produced strategy guarantees survival. For guaranteeing a non-zero chance of reaching T, another invariant is used. The invariant says that after each iteration of the repeat-loop, for each state s with $\boldsymbol{r}(s) \leq cap$, there exists a finite $\Sigma^{\boldsymbol{r}}$-compatible path starting in s and ending in a target state, where the agent's current resource level never drops below the value of $\boldsymbol{r}$

corresponding to its current state. Essentially, the chance of reaching a target state is not zero if $\boldsymbol{r}(s) \leq cap$. The fact that the current resource level does not drop below the corresponding value of $\boldsymbol{r}$ is not directly necessary for proving the desired result, but it is helpful for proving the invariant. With these invariants, it follows that the produced strategy $\Sigma^{\boldsymbol{r}}$ is T-positive $SafePR_T(s)$-safe in each state s. The details can be found in [3].

6 Implementation and evaluation

We investigated the following questions:

- Does the strategy generated from the counter selector produced by Algorithm 3, indeed prevent resource exhaustion and guarantee a non-zero chance of reaching a target state $t \in T$, if starting in $s \in S$ with $SafePR_T(s)$ units of resource?
- What is the effect of the number of reload states on the runtimes of the presented algorithms?

To answer these questions, we implemented[9] the presented algorithms in C++ using the open-source model checker "Storm" as a basis. Since the algorithms had also been implemented in Python by the authors of [3], we additionally checked that our implementation gives the same result for the vectors $MinInitCons, Safe,$ and $SafePR_T$ as the Python implementation.

6.1 Setup

6.1.1 Verifying the produced counter selectors

The counter selector which is produced by Algorithm 3 is not unique because in line 3 an arbitrary safe action is used. That is to say, we cannot check that the counter selector produced by our implementation of Algorithm 3 is correct by comparing it to the one produced by the implementation of the authors of [3], since it is possible for the two counter selectors to be different yet both correct.

Instead, we checked the correctness of the produced counter selector Σ as follows. The vector $SafePR_T$ and a corresponding counter selector Σ were computed for each CMDP $\mathcal{M}$ from Subsection 6.1.3. We transformed $\mathcal{M}$ into an MDP $\mathcal{M}'$ where one special state represents resource exhaustion and every other state represents a tuple (s, d) with s a state from $\mathcal{M}$ and d a resource level. In each state s' of $\mathcal{M}'$ the only outgoing transition is the one corresponding to the action that $\Sigma^{\boldsymbol{v}}$ would choose if the agent were in s with d units of resource, where s' represents (s, d). Here $\boldsymbol{v} := SafePR_T$ and $\Sigma^{\boldsymbol{v}}$ is the strategy generated from Σ. That means that for a state from $\mathcal{M}'$ representing the tuple (s, d), the only runs which are possible in $\mathcal{M}'$ correspond exactly to the runs in $Comp_{\mathcal{M}}(\Sigma^{\boldsymbol{v}}, s)$. In other words, it is no longer necessary to consult the strategy $\Sigma^{\boldsymbol{v}}$: its behaviour is encoded into the transitions of the MDP $\mathcal{M}'$.

Subsequently, we used an algorithm from [2, Chaper 10] that was implemented in Storm, to confirm that for each state s from $\mathcal{M}$, the agent could never reach the state representing resource exhaustion if it started in the state representing the

[9] https://github.com/arthuralsett/storm/tree/cmdp

tuple $(s, SafePR_T(s))$. This would show that $\Sigma^{\mathbf{v}}$ is $SafePR_T(s)$-safe in each $s \in S$. Furthermore, for the states representing $(s_i, SafePR_T(s_i))$ we used an algorithm implemented in Storm[10] to confirm that the probability of a run which visits a target state $t \in T$ is not zero. This would show that $\Sigma^{\mathbf{v}}$ is T-positive $SafePR_T(s)$-safe in each $s \in S$, assuming we already know it is $SafePR_T(s)$-safe.

6.1.2 Runtimes versus number of reload states

We investigated the influence of the number of reload states on the runtime of the algorithms. We applied the algorithms to CMDPs modelling grid worlds: the set of states corresponds to a 45×45 grid. The capacity was set to 5 and the number of target states to 102. All reload and target states are distributed randomly over the grid according to a uniform distribution. There are four actions: moving in the directions north, east, south, and west, respectively. Each action has a cost of 1. When the agent moves in a particular direction, the probability of moving one block is 0.9 and the probability of moving two blocks in that direction is 0.1.

However, what the exact values of these probabilities are, will not make a difference for the results of the algorithms as long as neither probability is zero and they sum to one. The reason is as follows. The minimal initial resource level d for which there is a strategy that is (T-positive) d-safe in a state s depends on which runs are in $Comp(\sigma, s)$ for each strategy σ. For a fixed strategy σ, the values of the aforementioned probabilities can only affect the contents of the set $Comp(\sigma, s)$ by changing whether specific elements of $(S \times A)^\omega$ are valid runs. But to determine whether an element $\rho \in (S \times A)^\omega$ is a valid run, it only matters whether for each subsequence $s_i a_i s_{i+1}$ we have $\Delta(s_i, a_i)(s_{i+1}) > 0$. Hence the results of the algorithms will not change if these probabilities do not change between being zero and non-zero. This is also why none of the algorithms consult the actual values of these probabilities, although they indirectly check whether they are non-zero by using $Succ$.

6.1.3 Comparing the implementations

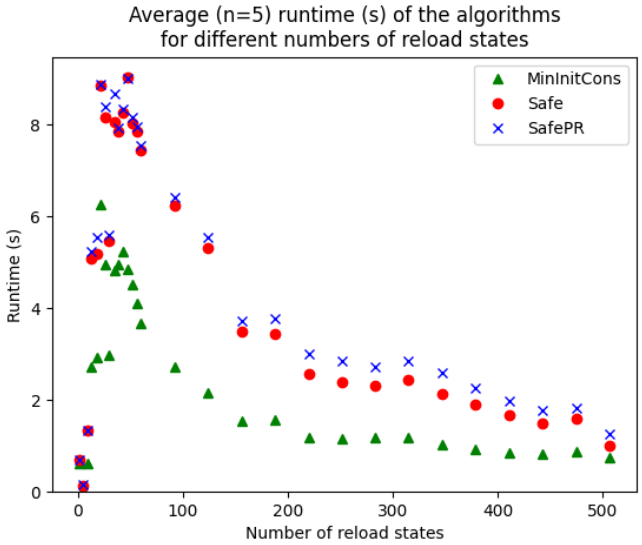
We checked that our C++ implementation gives the same results for the vectors $MinInitCons, Safe,$ and $SafePR_T$ as the Python implementation. We used CMDPs as in Subsection 6.1.2, except with different values. Both sets of algorithms were applied to CMDPs with $n \times n$ grids, $\lceil xn^2 \rceil$ reload states, $\lceil yn^2 \rceil$ target states, and a capacity of cap, where each combination of the following values was taken:

$$\begin{aligned} n &= 20, 30, 40, 50, \\ x &= 0.1, 0.2, 0.3, \\ y &= 0.05, 0.1, 0.15, \\ cap &= 3, 5, 7. \end{aligned}$$

The vector $MinInitCons$ is allowed to have finite values greater than cap in the Python implementation, but in our implementation such values are replaced by ∞. We performed our tests such that differences caused by this implementation detail were ignored since it is a small design choice and not a significant semantic issue.

[10] See [4].

Figure 5:



6.2 Results

The tests showed that for all tried CMDPs, the produced counter selectors were correct and our implementation agreed with the implementation by the authors of [3].

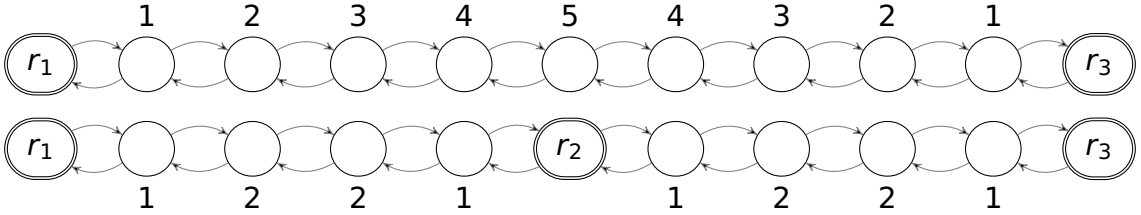
In Figure 5 the runtimes are plotted for various numbers of reload states, where for each input size the average runtimes of five different inputs was taken. After an initial spike, the runtimes for all three algorithms decrease as the number of reload states increases. We discuss possible causes for this. Algorithm 1, in the first few iterations of its loop, gives the correct values to states which are close to reload states. In general there can be more complex situations, such as when a state s has a possible transition to a reload state which is immediate (i.e. without intermediate states) but expensive, as well as a possible series of cheap transitions which lead to the same reload state. If the summed cost of the cheap transitions is less than the expensive transition, Algorithm 1 only gives the correct value to s in a later iteration of the loop. However, in the current setting this is not a concern since all actions have the same cost in each state.

In each subsequent iteration, Algorithm 1 assigns the correct value to states which are a little further removed from reload states. This leads to chains of states starting at reload states where the states have increasing required initial resource levels as the chain moves away from the reload state where it started.

Algorithm 1 terminates when there are no more changes to be made to the vector it computes. Therefore the runtime depends on how quickly this fixed point is reached. If there are more reload states, it is more likely that chains meet eachother as they move away from reload states. When two chains meet eachother, the individual chains do not go up from zero to cap, but instead they end at whatever value they had when they met. This could lead to fewer iterations and thus a lower runtime. In Figure 6 a simplified example is given to illustrate the idea.

One reason the runtime of Algorithm 2 decreases with more reload states, is that

Figure 6: the numbers above and below the states indicate *MinInitCons* values.



it uses Algorithm 1. Another reason could be that fewer iterations are needed because fewer reload states need to be removed from *Rel*. First, if there are few reload states then the reload states are sparse in the grid and it will occur more often that reload states r_i are isolated from other reload states and need to be removed from *Rel* because it is not possible to guarantee reaching another reload state from *Rel* from the states r_i. If there are more of those "bad" reload states, it will be more likely that they form a chain where they have to be removed one by one. See Example 1. Clearly, chains of reload states that have to be removed one by one[11] will lead to more iterations being required in Algorithm 2 and thus a longer runtime.

The reason the runtime of Algorithm 3 decreases with the number of reload states is likely mostly caused by it using Algorithm 2, given that the runtimes of Algorithms 2 and 3 are so close. The difference between their runtimes is too small compared to the absolute runtime, for a statement to be made about whether the number of reload states influences the runtime of Algorithm 3 through the part of Algorithm 3 that does not use Algorithm 2.

7 Conclusion

In this thesis, we discussed a model that captures planning problems for systems with rechargable batteries where the results of actions are non-deterministic. We formally specified two problem statements related to preventing resource exhaustion and the agent reaching a target state. The first problem, called *safety*, is to find for each state the minimal initial resource level such that resource exhaustion can be prevented, as well as to find a single[12] strategy that indeed prevents resource exhaustion when starting in an arbitrary state, if given the minimal initial resource level corresponding to that state. The second problem, called *positive reachability*, is similar to the first, except the goal is not simply to prevent resource exhaustion, but additionally for the agent to reach a target state with a non-zero probability.

We presented algorithms to solve these two problems. For the safety problem, constructing a strategy was done by fixing a *safe action* in each state. For the positive reachability problem, a strategy was constructed by assigning to each state a partial mapping ϕ from resource levels to actions. At any point in time, the agent's next action is $\phi(d)$, where d is the maximum resource level such that d is not greater than the agent's current resource level and such that $\phi(d)$ is defined. When no such d

[11] Or possibly a few at a time, but not all at once.

[12] As opposed to a separate strategy for each state.

exists, a globally fixed default action is used. Correctness as well as polynomial time complexity were shown for the algorithms.

We re-implemented the presented algorithms in C++ and improved our confidence that our implementation was correct by comparing the output to that of the implementation by the authors of [3]. Furthermore, we verified the strategies our implementation produced for the positive reachability problem by encoding their behaviour into an MDP and using existing techniques to verify that they satisfy the desired qualities.

Finally, we investigated the influence of the number of reload states on the runtimes of the algorithms and came up with possible explanations for the fact that a higher number of reload states leads to a lower runtime. For Algorithm 1, we speculate that chains of states leading away from reload states meet more often, leading to fewer iterations being required. For Algorithm 2, besides the fact that it uses Algorithm 1, a possible cause is that there are fewer chains of "bad" reload states which need to be removed one by one. The main reason for Algorithm 3 is likely that it uses Algorithm 2.

References

[1] Robert B. Ash and Catherine A. Doléans-Dade. *Probability and Measure Theory*. Harcourt/Academic Press, 2000.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] František Blahoudek et al. "Qualitative Controller Synthesis for Consumption Markov Decision Processes". In: (2020). DOI: 10.48550/ARXIV.2005.07227. URL: https://arxiv.org/abs/2005.07227.

[4] Christian Hensel et al. "The probabilistic model checker Storm". In: *Int. J. Softw. Tools Technol. Transf.* 24.4 (2022), pp. 589–610. DOI: 10.1007/s10009-021-00633-z. URL: https://doi.org/10.1007/s10009-021-00633-z.