Markov Decision Processes I

Table of Contents

[Non-Deterministic Search 2](#_Toc138878344)

[Markov Decision Process 3](#_Toc138878345)

[Policies 4](#_Toc138878346)

[MDP Search Trees 5](#_Toc138878347)

[Discounting 6](#_Toc138878348)

[Stationary Preferences 7](#_Toc138878349)

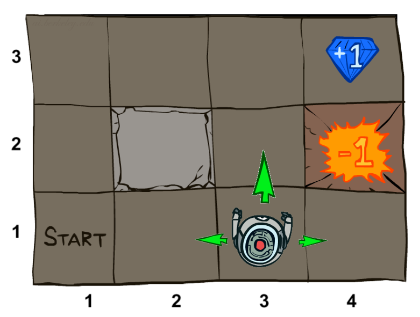
[Infinite Utilities 7](#_Toc138878350)

[Solving MDPs 8](#_Toc138878351)

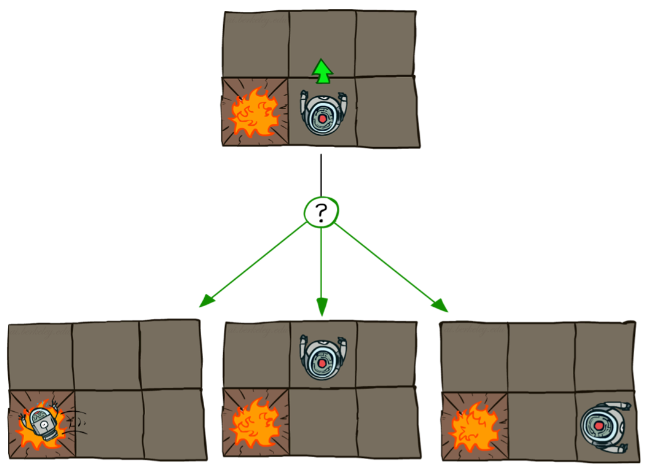
## Non-Deterministic Search

So far, we have dealt with search problems that were **deterministic**, i.e., we knew what the outcome of a set of actions would be for certain. The alternative to this is **non-deterministic** search, where the outcome of an action is unknown, or there is a possibility of the action failing. We touched upon non-deterministic search problems when studying **expectimax search**, but we will be going into more details now.

Consider that we have a **grid world** where an agent can go in one of four directions from any position, except if there is a wall in that direction. The goal is to reach a gem, which will give a large reward. There are also lava pits, which have a large negative reward. At each step, the agent receives a **living reward**, which could be positive (to encourage the agent to prolong the game) or negative (to encourage quickly finishing the game). Obviously, the goal is to maximize the reward.



The trick is, the agent’s actions have a possibility of failing. If the agent wants to go in a certain direction, there is a 10% change that they will go left instead, and a 10% chance that they will go right instead. If there is a wall in the direction the agent is trying to move in, then the agent stays where it is. Thus, the grid world is **stochastic**.



## Markov Decision Process

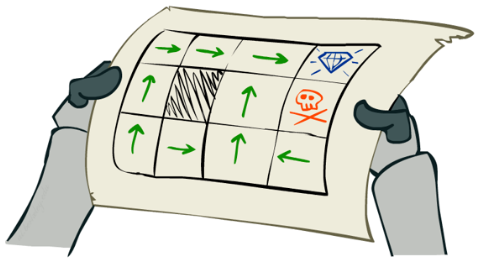
A **Markov Decision Process** (MDP) is defined by:

* A set of states,
* A set of actions,
* A transition function, , also called the model or dynamics, which simply gives the probability of transitioning to from by taking the action ,
* A reward function, , which gives a reward when taking an action at state . This is the living reward. Sometimes, this value could just be or if the reward is the same regardless of the action, current state or new state.
* A start state
* Maybe a terminal state

MDPs are non-deterministic search problems. One way to solve them is by using expectimax search, but we will also be exploring other mechanisms. In an MDP, the results of an action depend only on the current state.

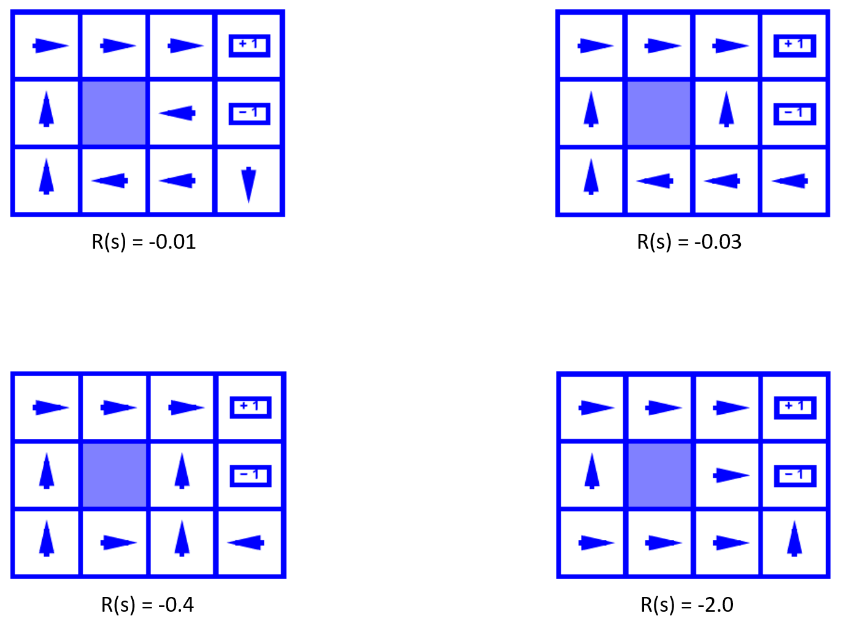
## Policies

In deterministic search, our goal was to create a plan, which was a sequence of actions to take in order to reach the goal optimally. However, we cannot use this for non-deterministic problems. What if one of the actions in our plan fails and we end up in a state that we should never have reached had we followed the plan? Thus, for MDPs, we need to instead tell the agent what action it should take **at every state** in order to reach the goal optimally. This is called a **policy** (). An optimal policy, like an optimal plan, **maximizes the expected utility**.



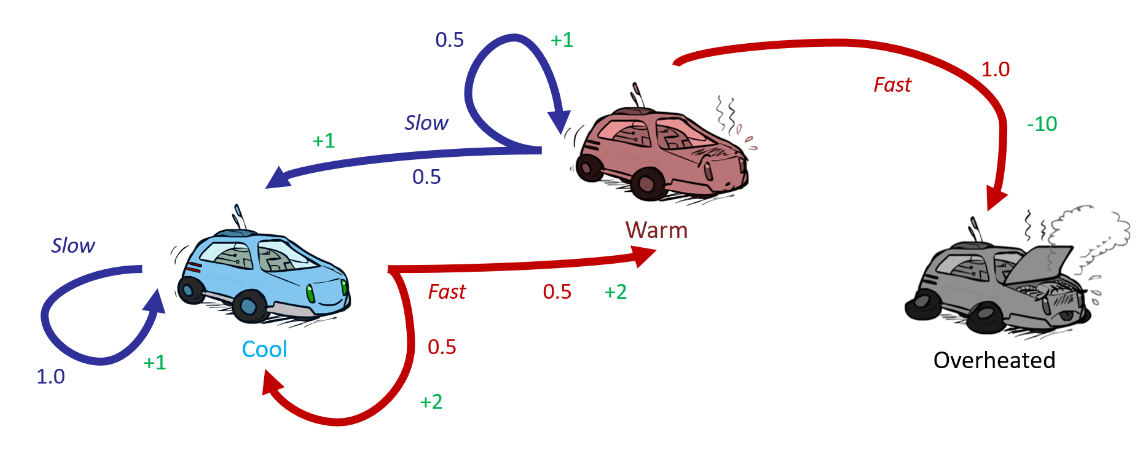
One possibility is to explicitly define the policy beforehand, i.e., we denote what action to take at each stage. An agent following such a policy would be a **reflex agent**. However, this is not always possible, especially for large trees. Expectimax does not compute entire policies, but rather computes what action to take for a single state only.

One interesting think to note that we will come back to later is that the optimal policy changes based on the living reward. This is because a longer, less risky path may be acceptable when the living reward is a small negative value, but a shorter, riskier path becomes acceptable if the living reward is a large negative value, because playing safe stops being an option.

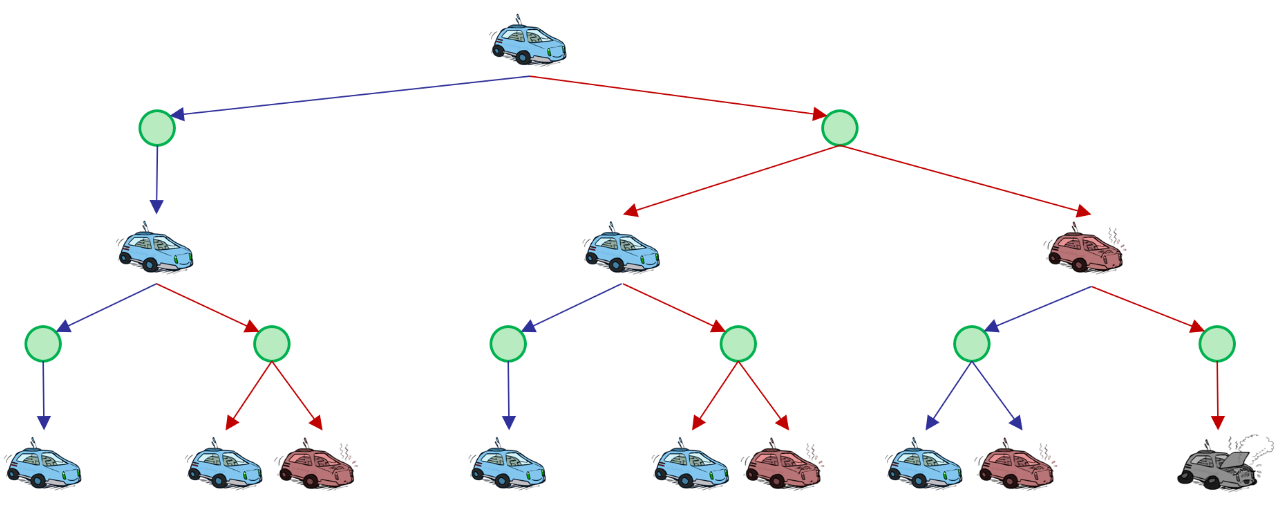


## MDP Search Trees

Suppose we have a car that we can choose to drive fast or slow, with driving fast being the desired situation. This car can be in one of three states, as shown below:



We can create a search tree from the example graph that would look like this:



This is an **MDP Search Tree**, which is a like an expectimax search tree. From each state, the action we choose to take takes us to a **chance node**, called a **q-state**. This node exists because there is a possibility of the action we took failing and something other than what we intended occurring. For example, when we are in the cool state, if we choose to take the action of driving fast, we might end up in either the cool state or the warm state.

## Discounting

Generally, our agents should aim towards getting a higher reward as quickly as possible. Since we are now dealing with problems that are **sequences of rewards** instead of single rewards, this can take form of choosing [2, 3, 4] over [1, 2, 2] and [1, 0, 0] over [0, 0, 1] respectively.

To factor this into our calculations, we use the concept of **discounting**. We pick a variable, , which is between 0 and 1, and for each step we take, the final reward is multiplied by . Thus, the final reward keeps decreasing exponentially the longer we take to reach it. In addition, this mechanism also helps our algorithms converge.

For example, if we have , the reward sequence has the utility while the reward sequence has the utility . Thus, .

## Stationary Preferences

**Stationary Preferences** are ones which do not change when the same reward is added to each of them at the same position, i.e.,

For stationary preferences, there are only two ways to define utilities:

1. Additive Utility,
2. Discounted Utility,

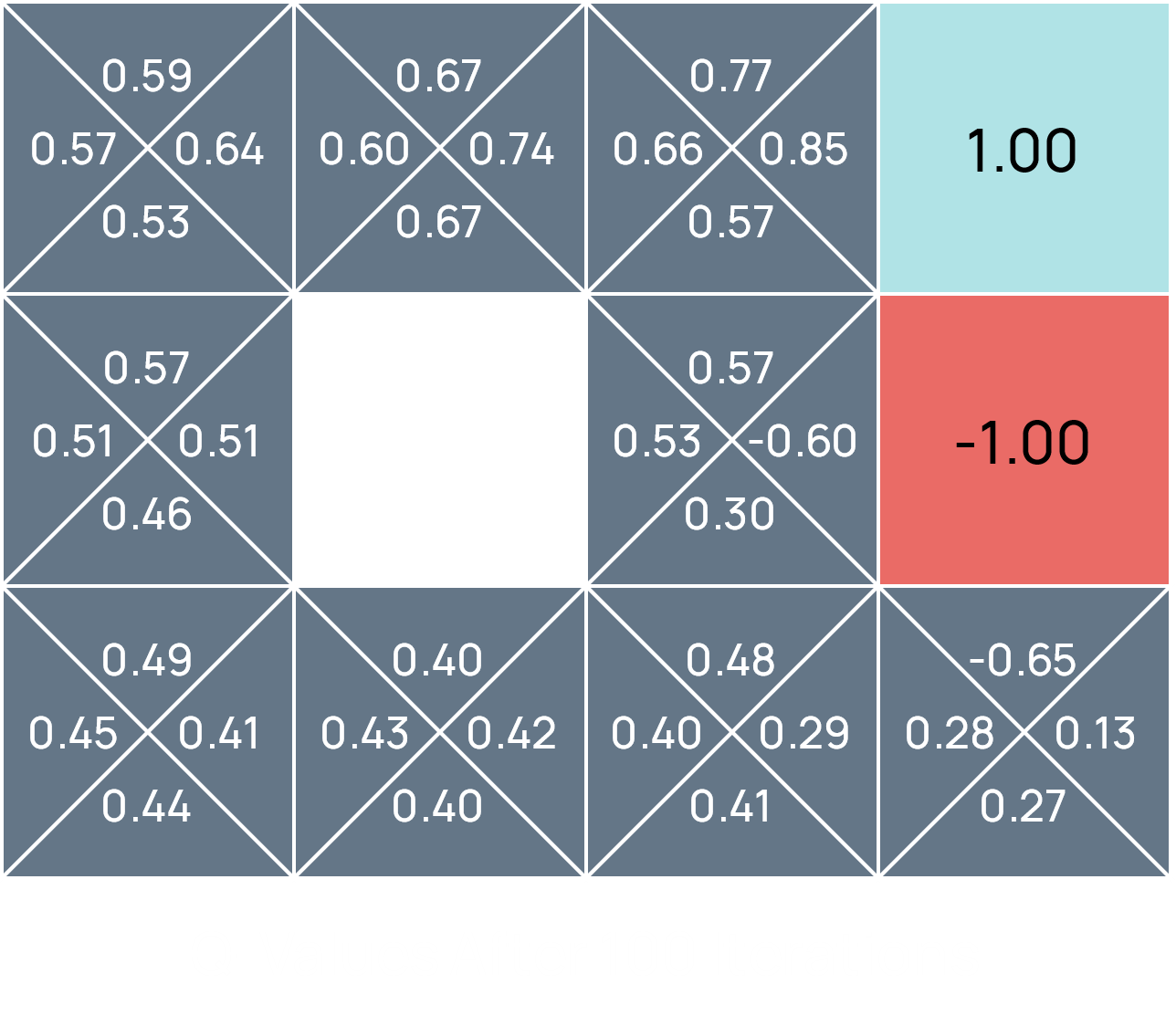
## Infinite Utilities

There are certain situations that have no terminating states. Notice that the example we saw earlier is one of these situations. In such case, there are a few ways we can handle the termination.

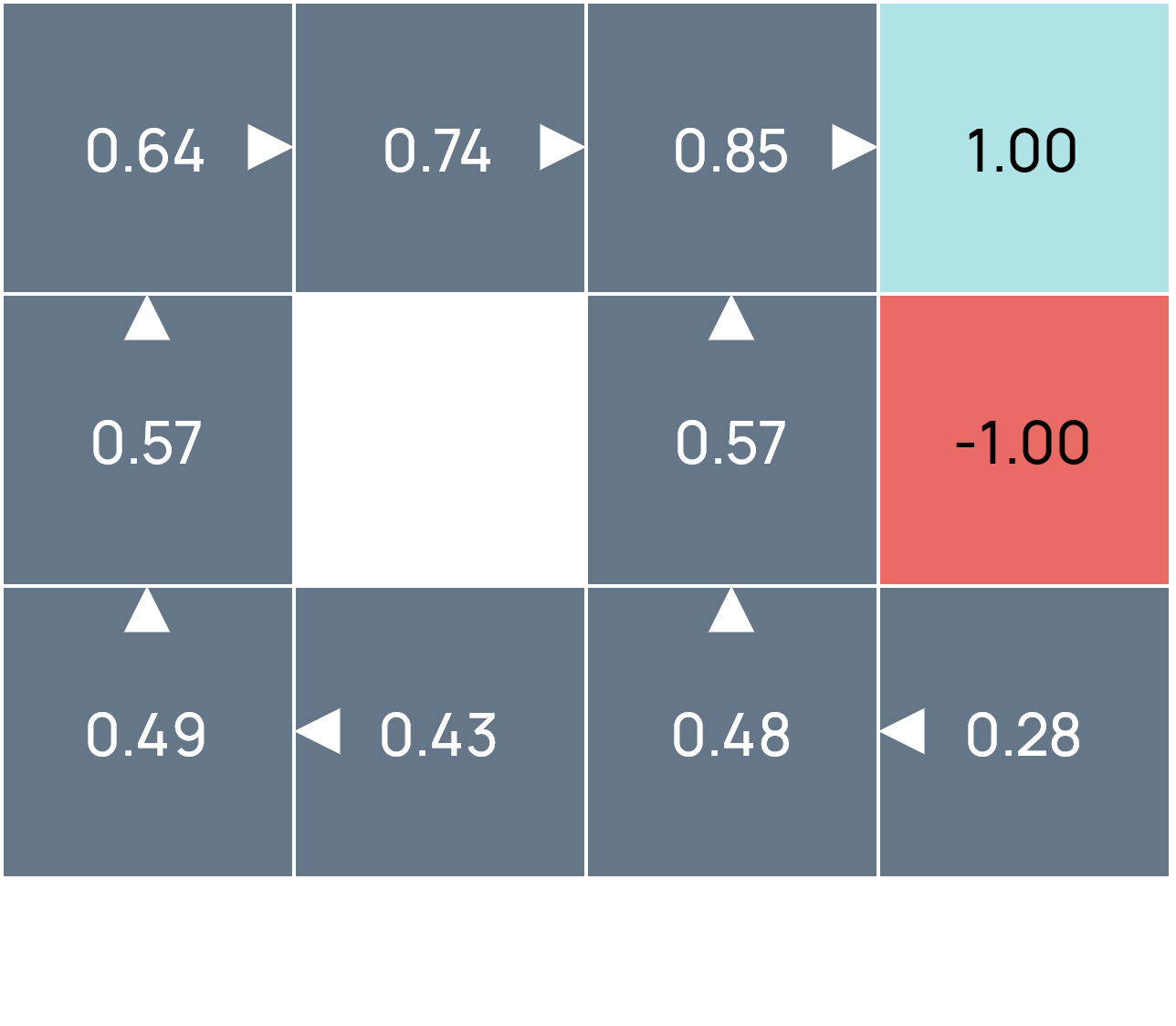
1. **Finite Horizon** – This is like depth-limited search, meaning we terminate searches after steps. The problem with this is that it gives us non-stationary policies, i.e., the policies are dependent on the amount of time left. A short depth gives us policies that are riskier without actually needing to be.
2. **Discounting** – Using a value in the range of 0 to 1 ensures that the reward values get infinitely smaller and eventually converge. There is a geometric backing to this, .
3. **Absorbing State** – We could also include a guarantee that for every policy, a terminal state will eventually be reached, such as the overheated state in our example. This is less something we can add and more an inherent part of the problem itself, which brings the argument that problems that have an absolute state are not infinitely long in the first place.

## Solving MDPs

The **expected utility of a q-state** , i.e., the chance node encountered after having started at and taken the action , and then acting optimally, is denoted as . Notice how we now have four values in each state, since the values are basically the expected utility if we take a specific action at a specific state.



The maximum value at any state is the **expected utility** of that state, . This value is the value we will get if we were to move optimally. It is not the actual value we will end up with if we play. The expected utility in turn dictates which action is the optimal action, .



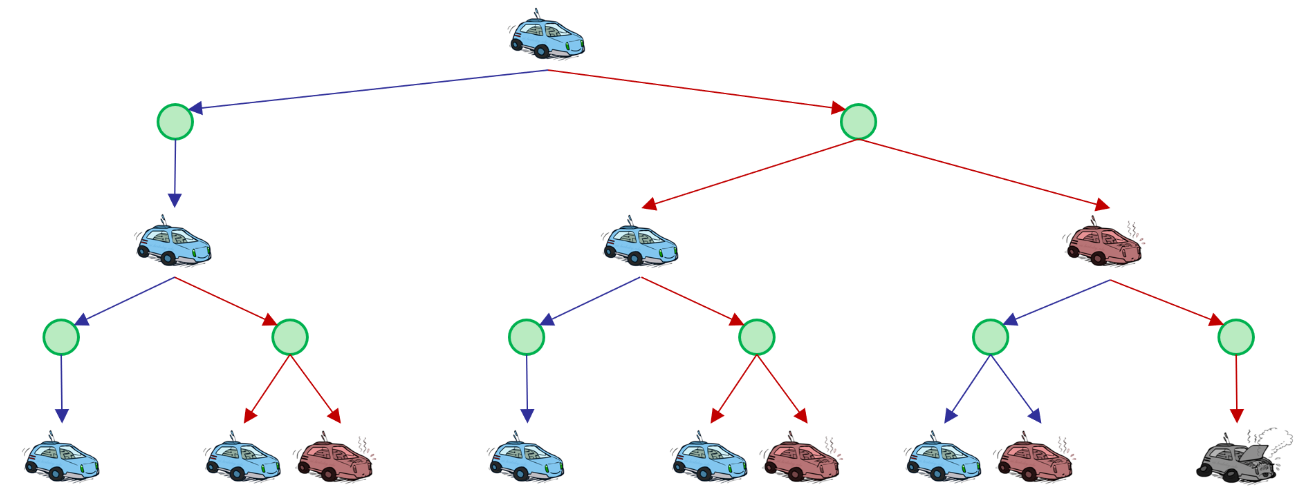
To calculate values, we use the following equations:

Thus, can be re-written as:

This is a very famous equation in the field of artificial intelligence, called the **Bellman Equation**.

However, this equation shows clearly that we are entering an infinite loop. How do we know when to stop?

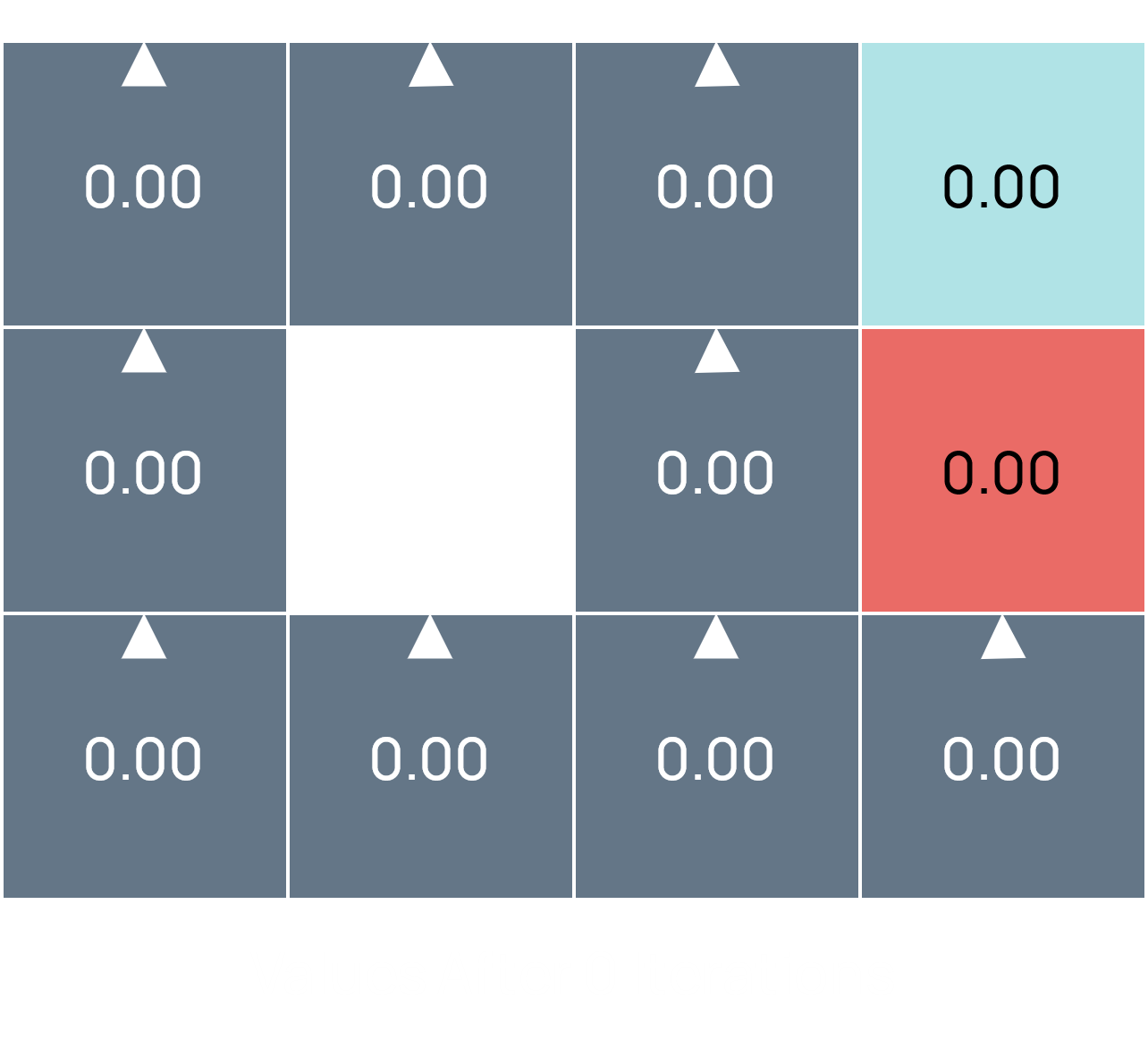
Let’s go back to the MDP tree for the racing car example for a moment.



There are two things we need to note about using such MDP trees. Firstly, we are repeating lots of states. At depth 2 for example, we are repeating the cool state twice. Since these two states are the same state and at the same depth, their values will also be the same. To avoid this repetition, we can use **memoization**. From here on, will denote the value of state at depth .

The second issue is the infinite length. To deal with this, we use **time-limited values**. Thus, will denote the optimal value of if the game were to end in moves.

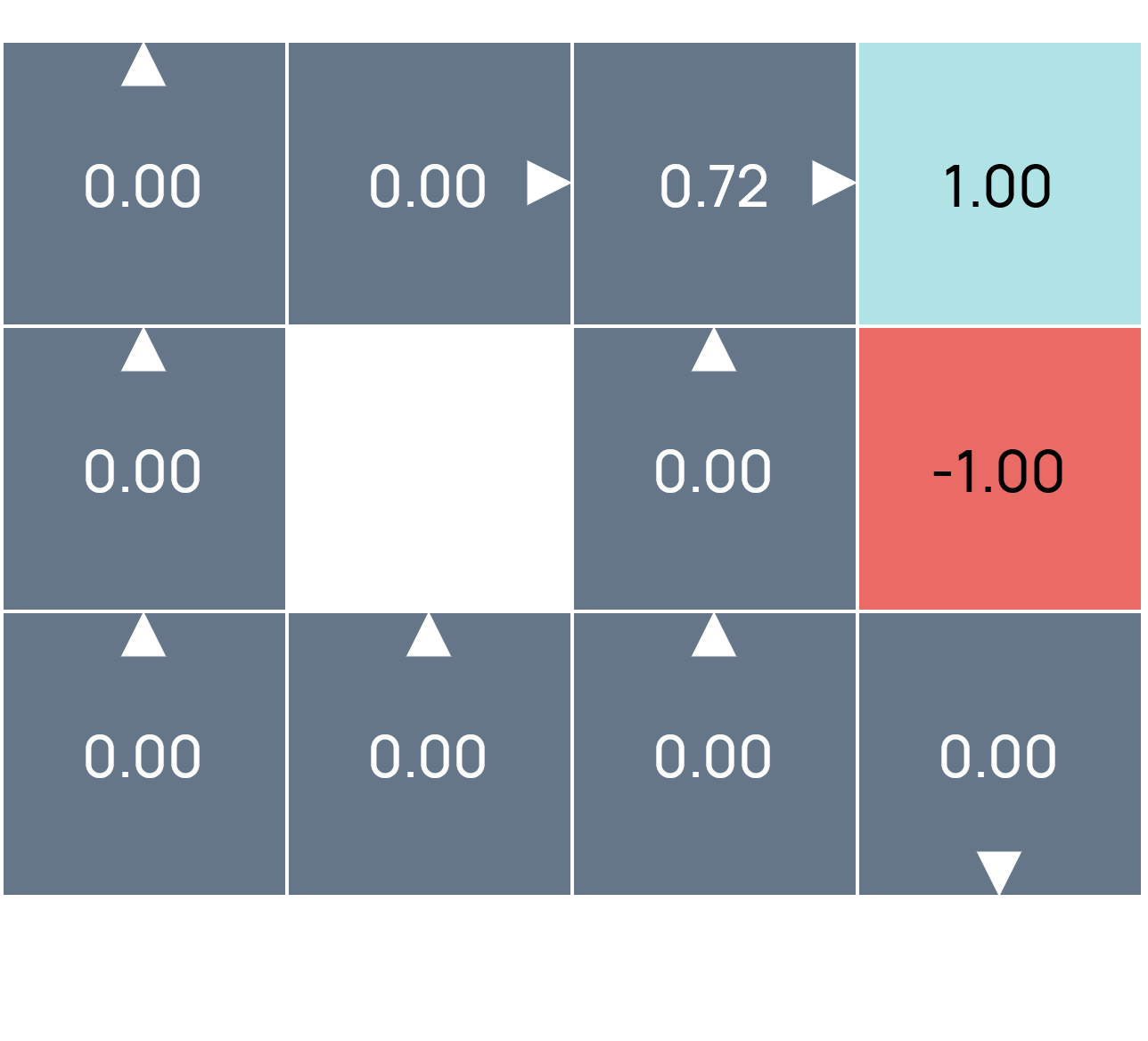
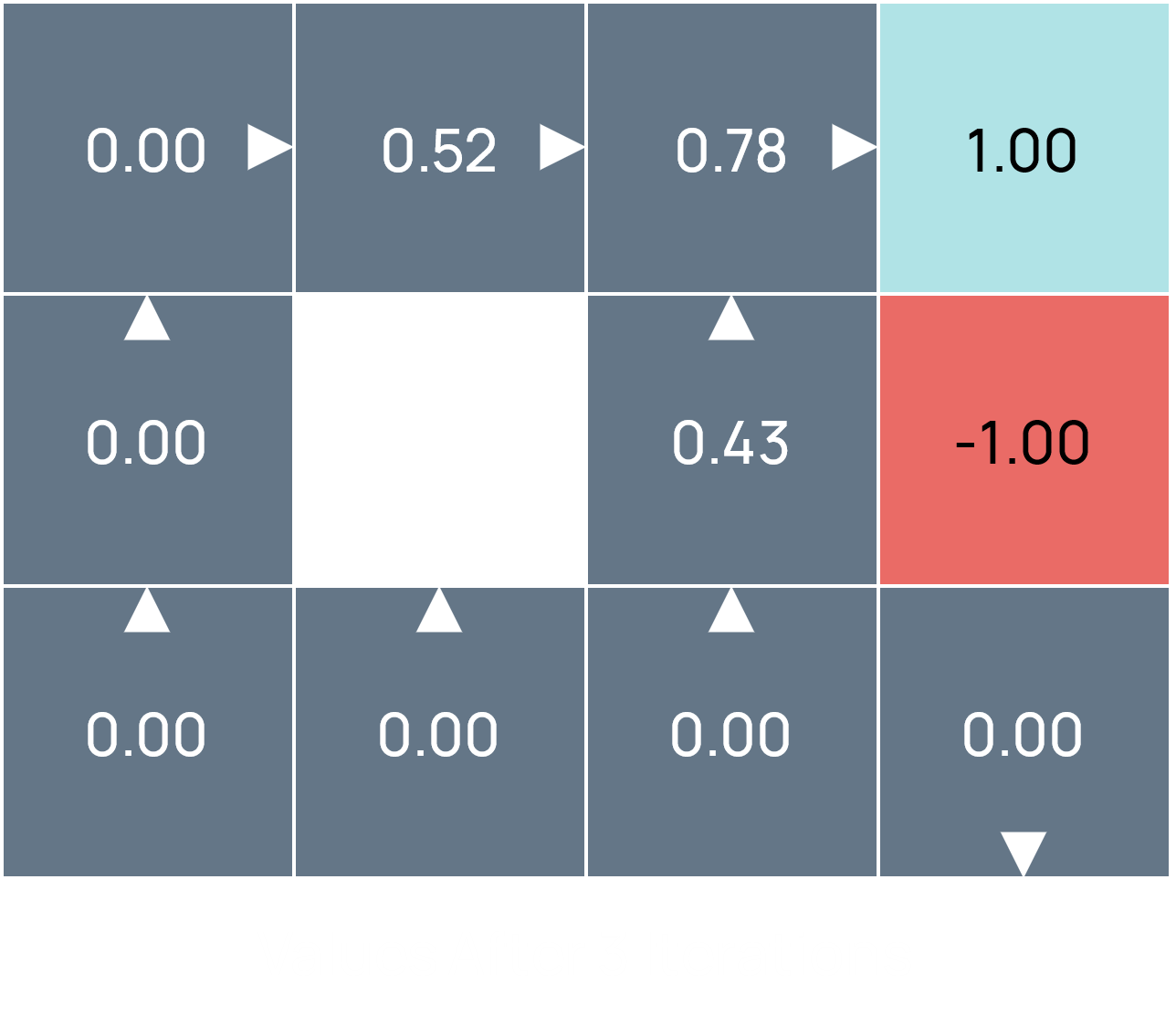
At for example, there are no moves we can make. Even if we are at the terminal state, we need one move to exit the game, which we cannot make. Thus, ever value is 0.

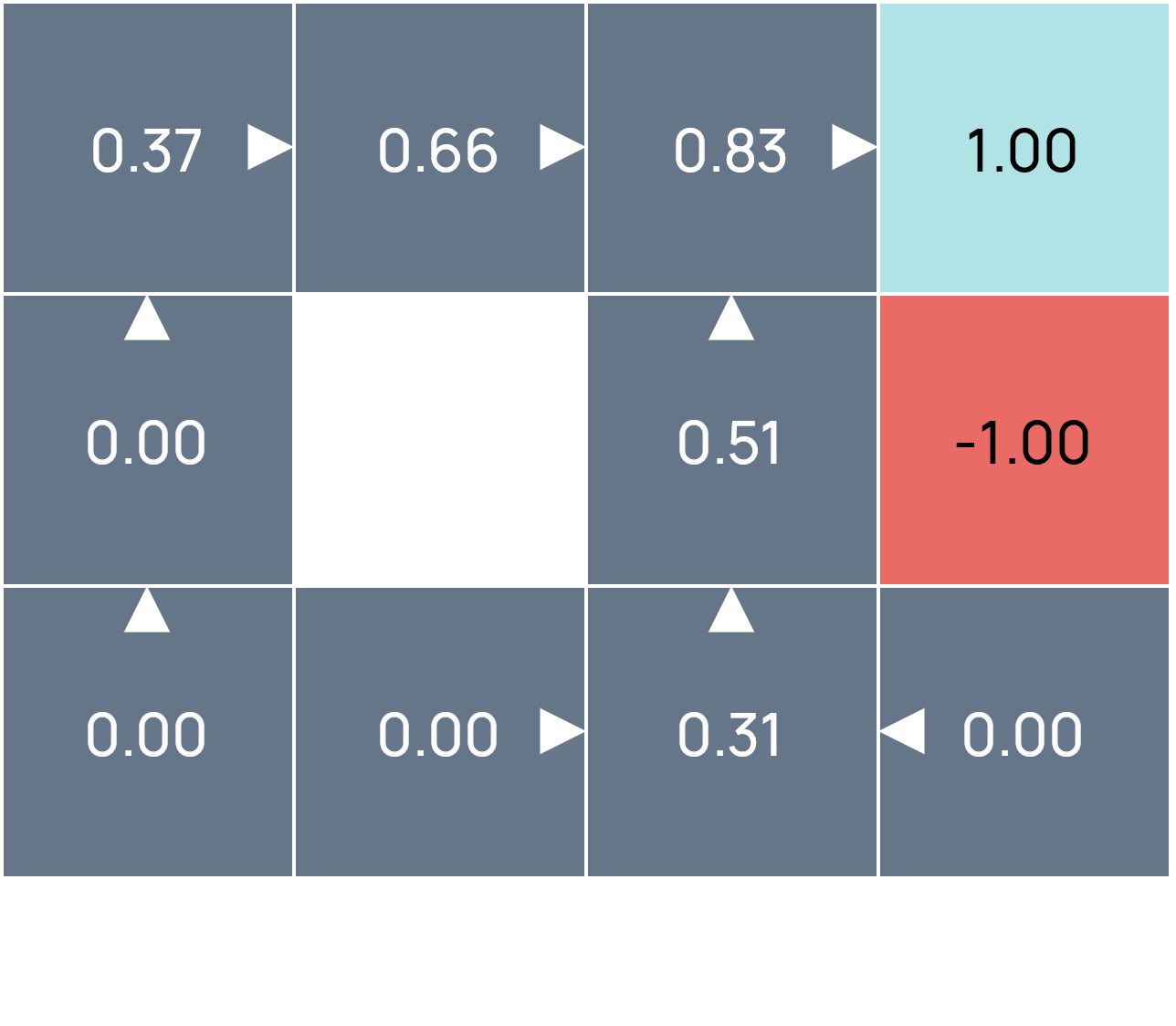
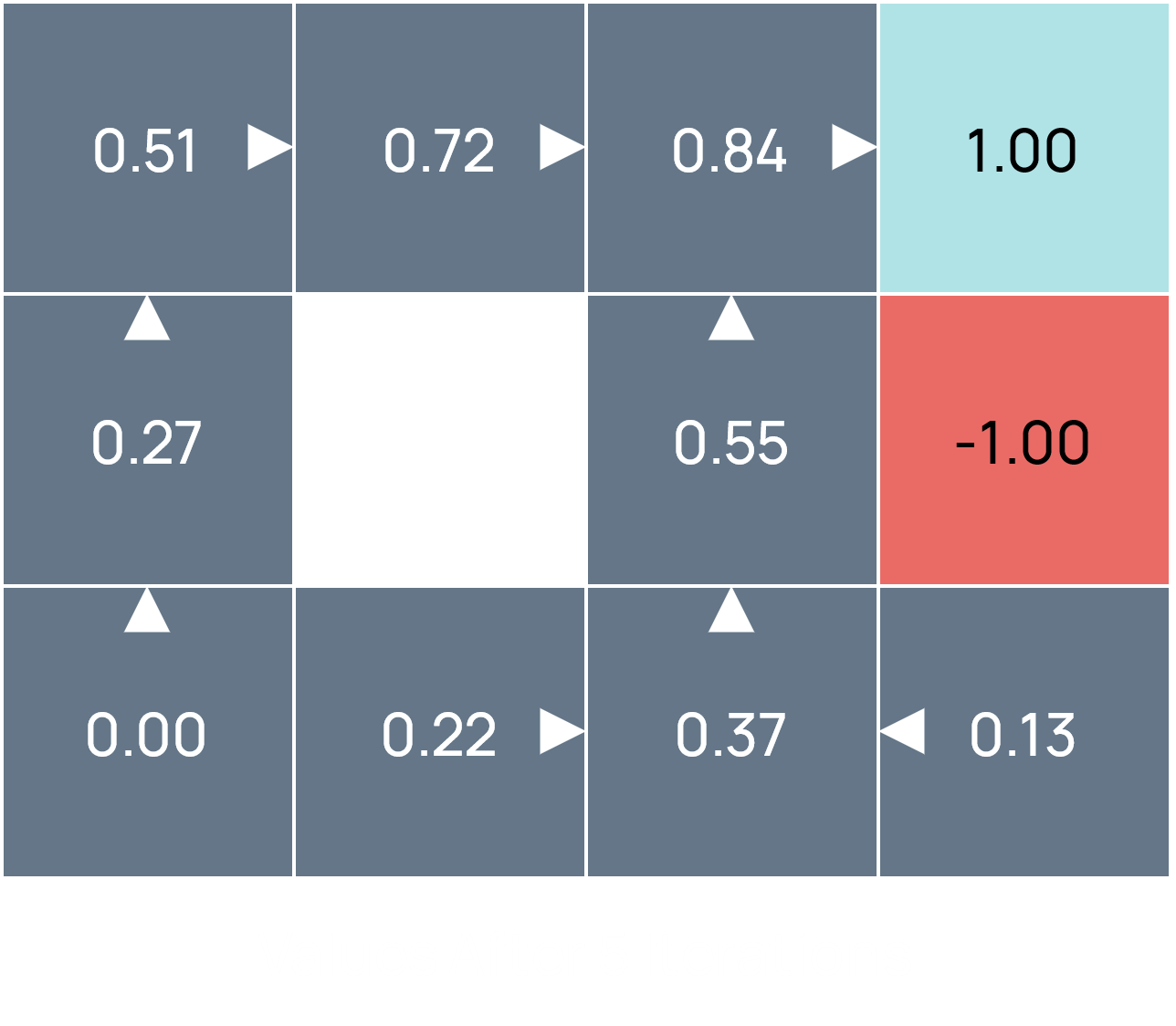


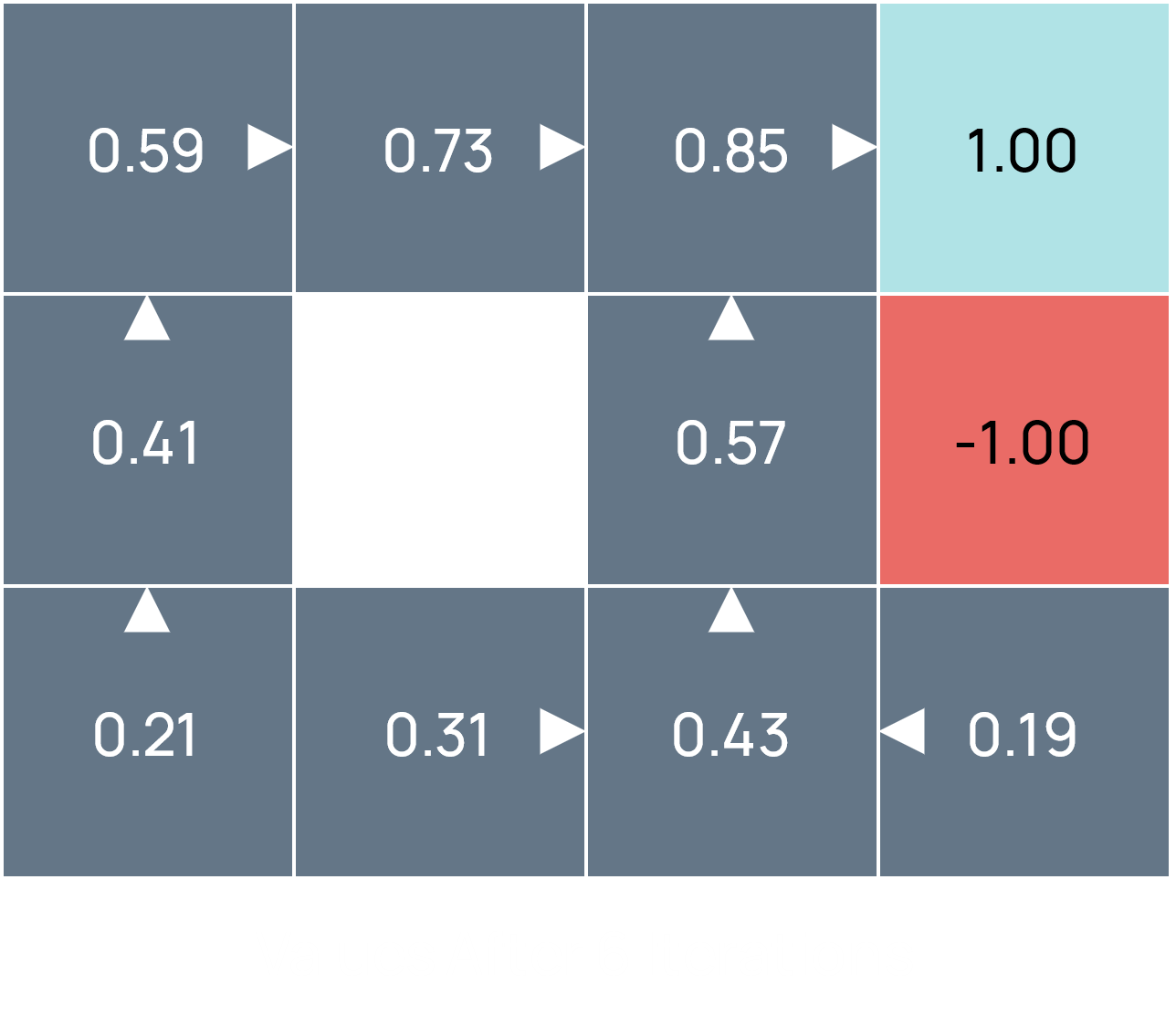
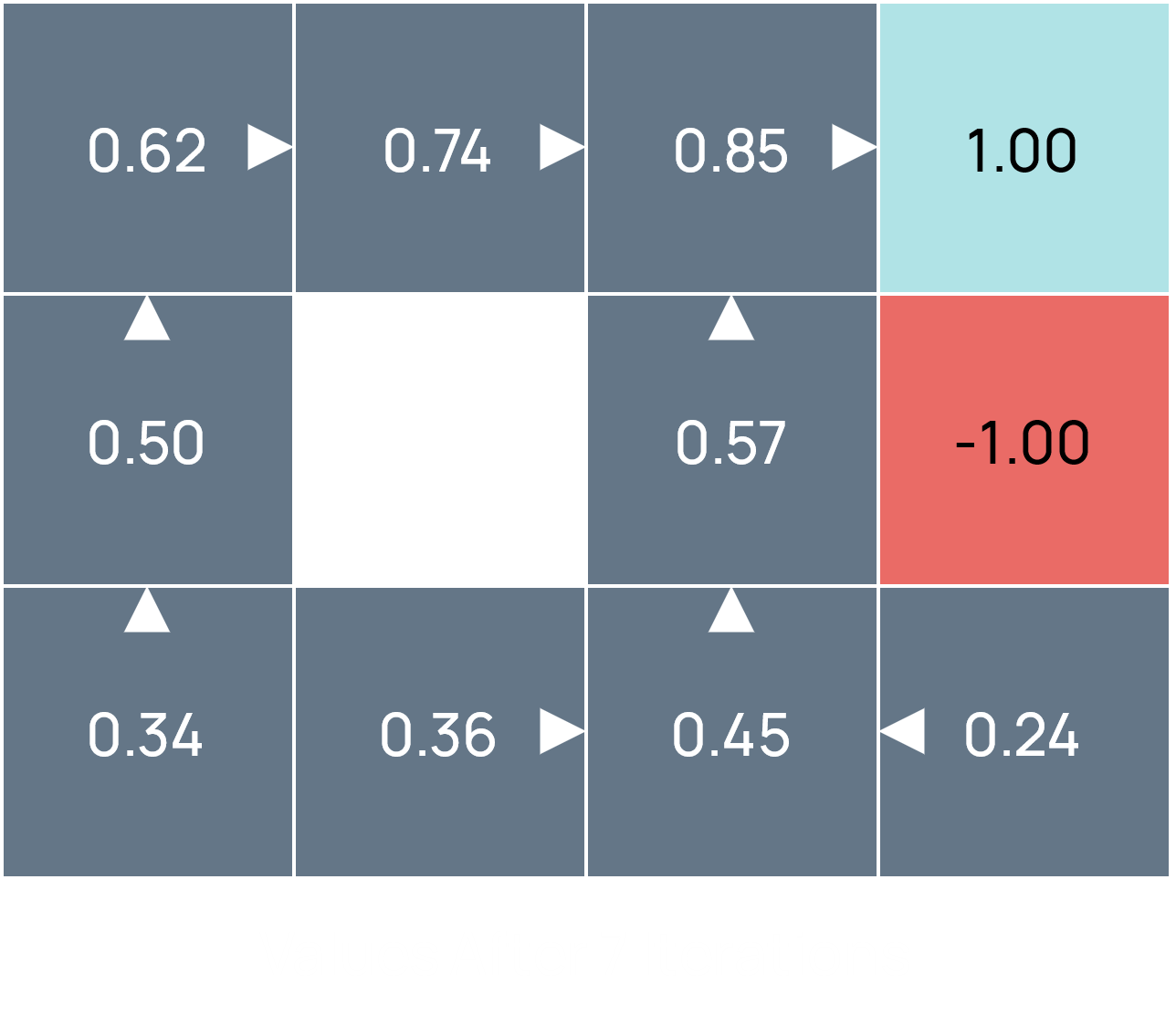
At , we have the option to exit at the terminal states, which gives the terminal states values, but no other state.

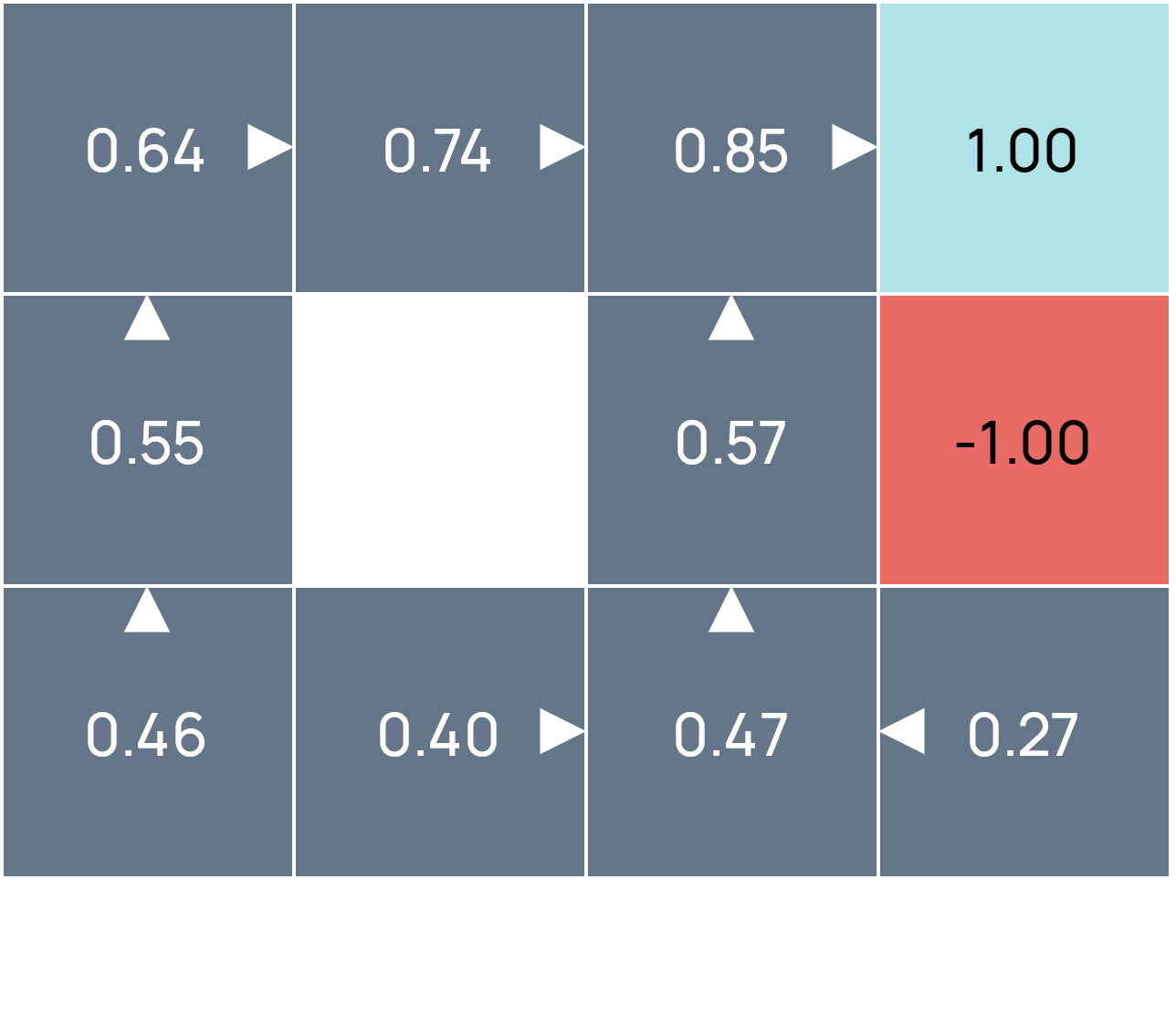


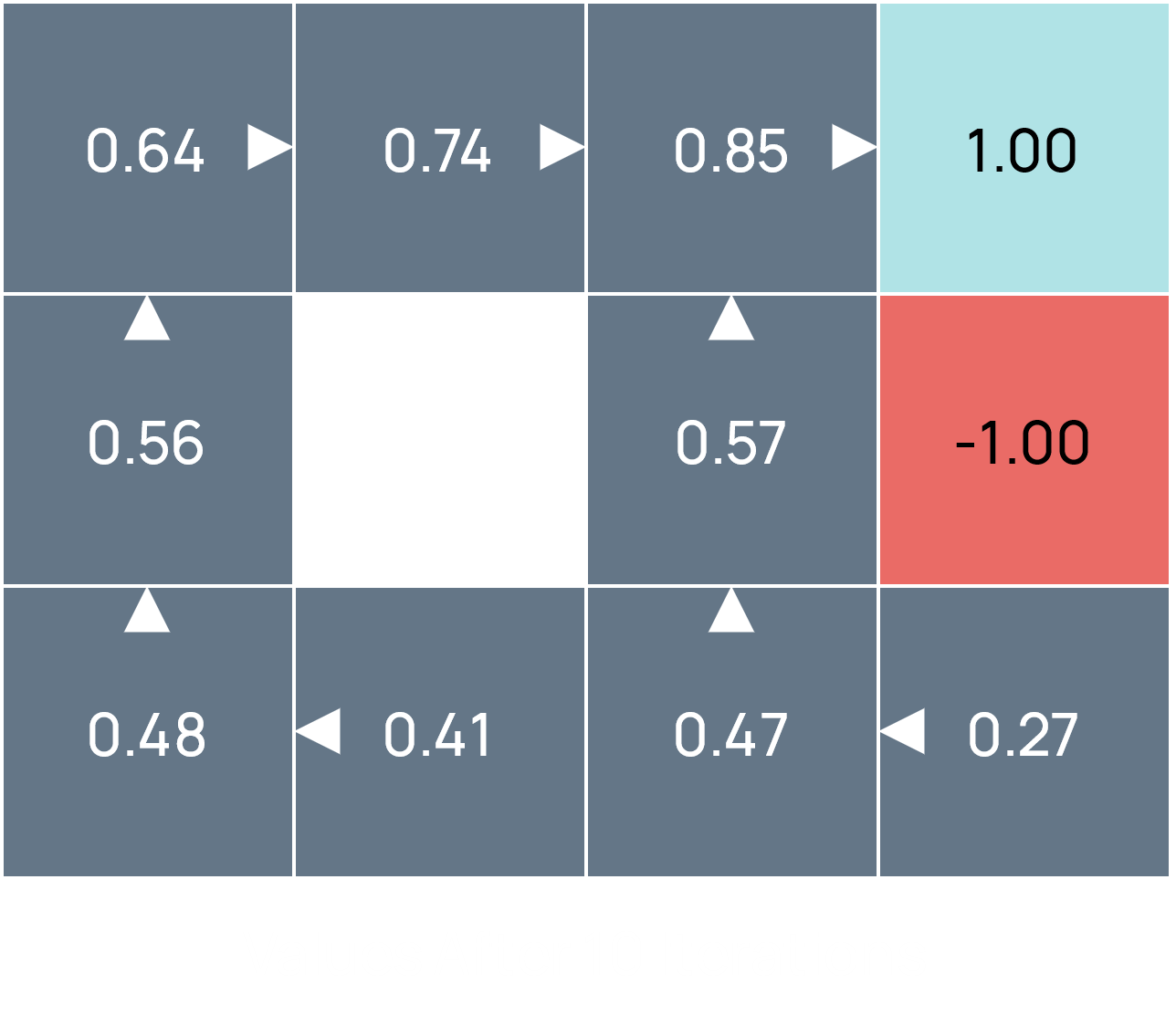
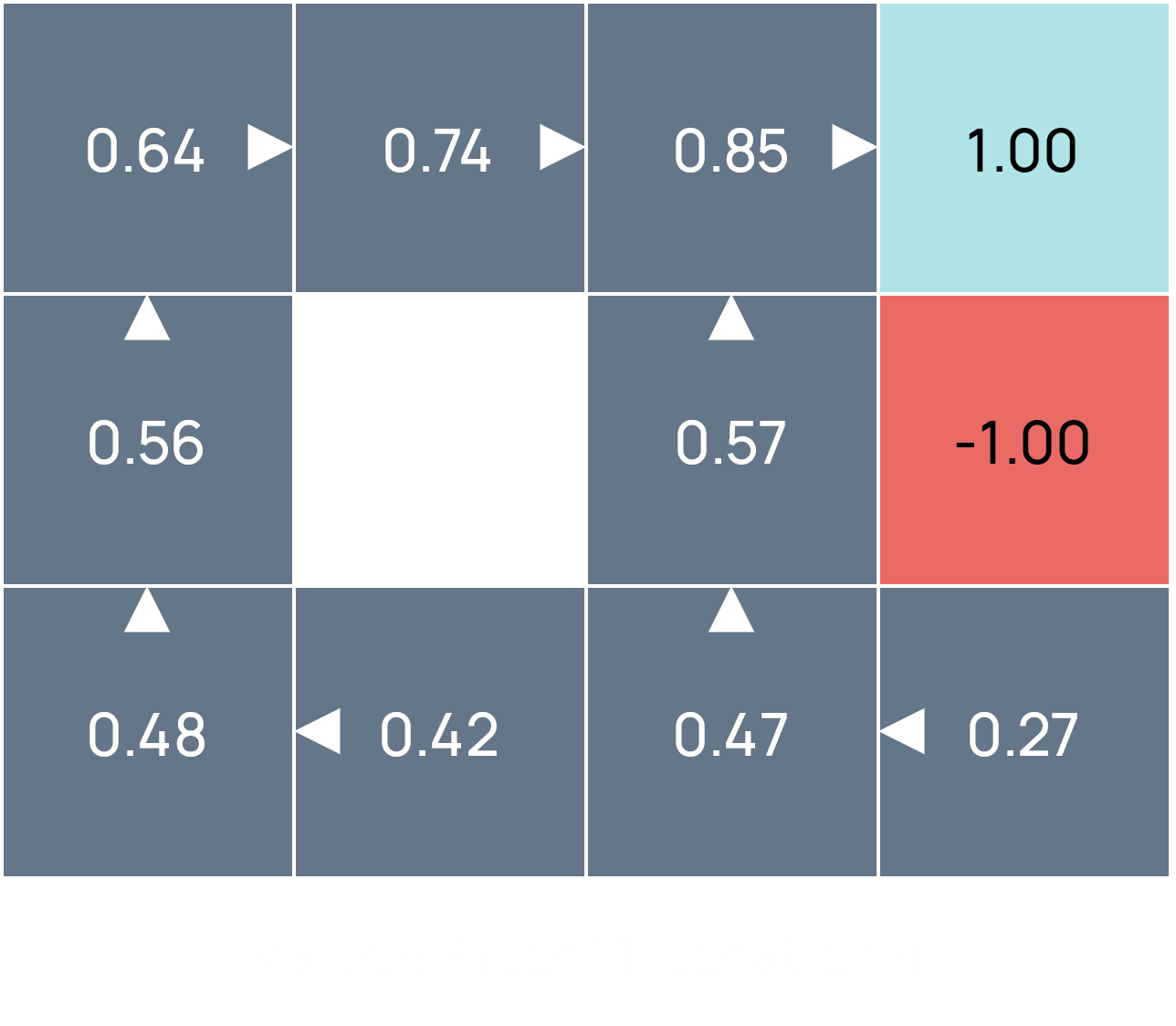
As we keep increasing the value of , the values of the other states will begin to appear and optimize themselves.

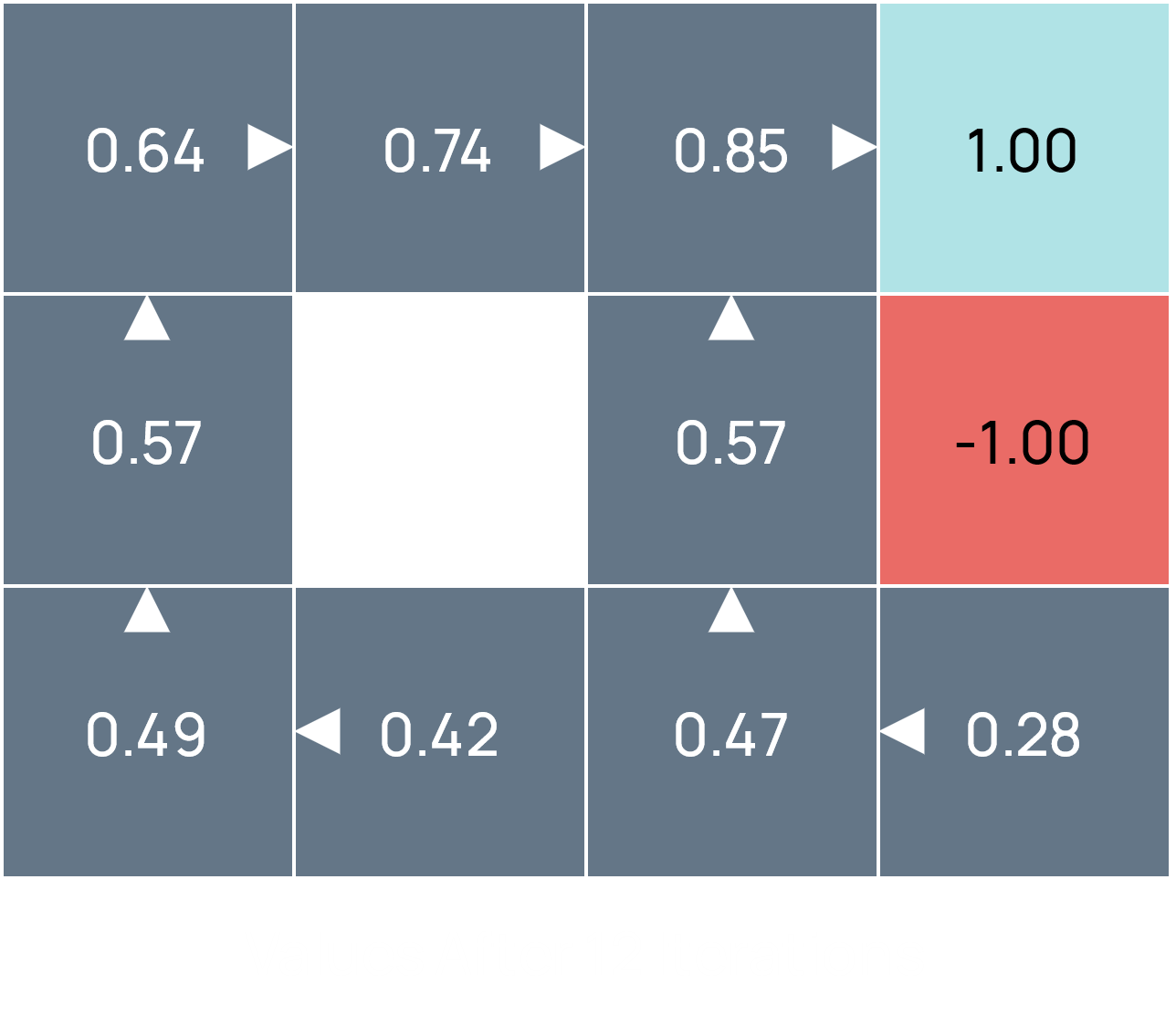
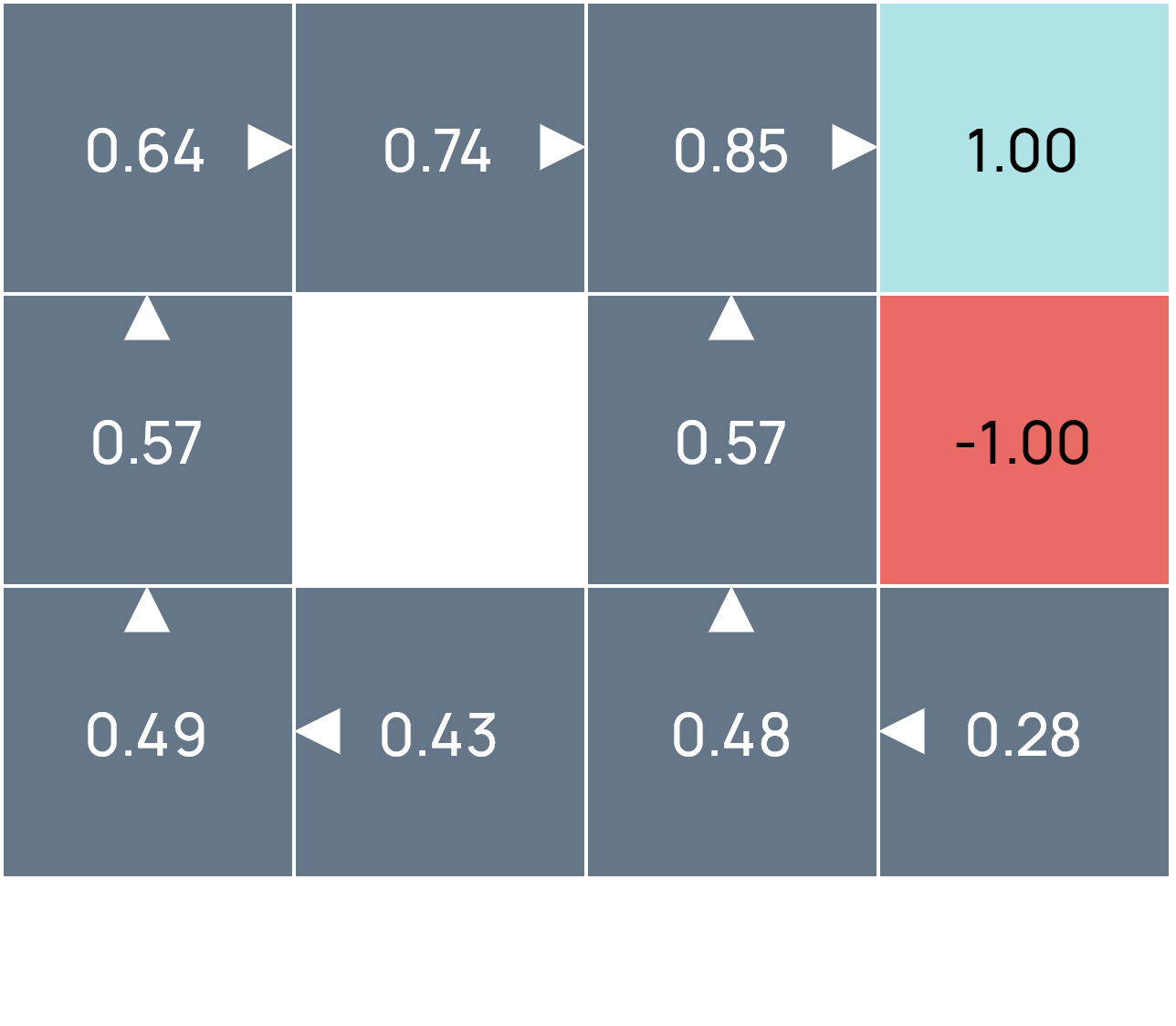
 

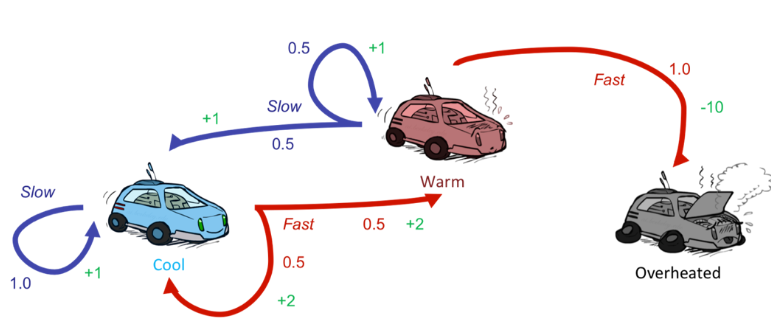
 

Notice that by the 12th iteration, the values have converged.

The process of increasing values of until convergence is called **value iteration**. This mechanism has a complexity of per iteration, since theoretically, it is possible to take every action at every state and reach every other state.

In value iteration, the policy converges long before the values do. This information will be useful later.

The actual process of calculating values is best understood using a different example.



For the time being, assume that there is no discount, i.e., .

For , all three states will have the value .

For ,

* For the cool state:
  + Going slow gives the value
  + Going fast gives the value
  + Thus,
* For the warm state:
  + Going slow gives the value
  + Going fast gives the value
  + Thus,
* For the overheated state, we cannot make any moves, so .

For ,

* For the cool state:
  + Going slow gives the value
  + Going fast gives the value
  + Thus,
* For the warm state:
  + Going slow gives the value
  + Going fast gives the value
  + Thus,
* For the overheated state, we cannot make any moves, so .

