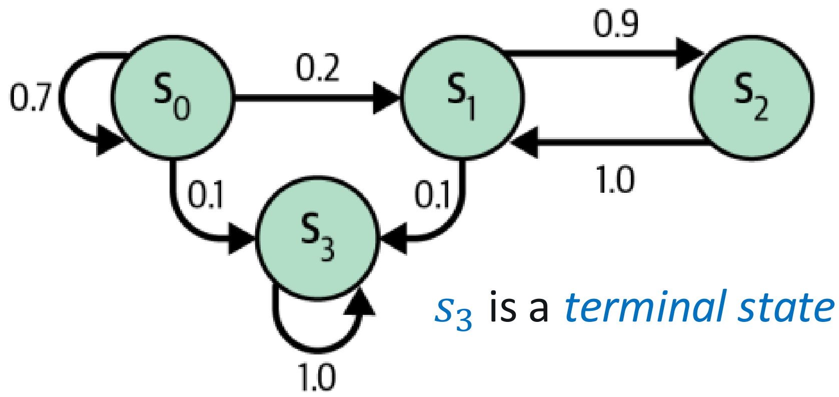
5.
Markov Decision Processes

Markov Chains

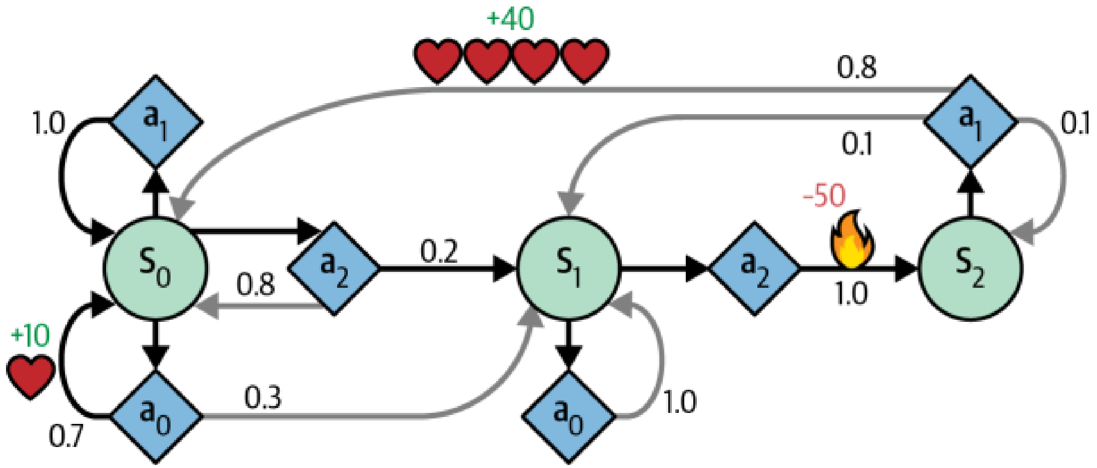
- *Markov chains:* stochastic processes with no memory.
  - Such a process has a fixed number of states, and it randomly evolves from one state to another at each step.
  - No memory: the probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s, s'), not on past states.



s_3 is a *terminal state*

Markov Decision Process

- Markov decision process (MDP): similar to Markov chains, but at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action.
  - Some state transitions return reward (positive or negative), and the agent's goal is to find a policy that will maximize reward over time.



Bellman Optimality Equation

- Bellman found a way to estimate the *optimal state value* of any state s, noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches the state, assuming it acts optimally.
- If the agent acts optimally, then *Bellman optimality equation* applies:

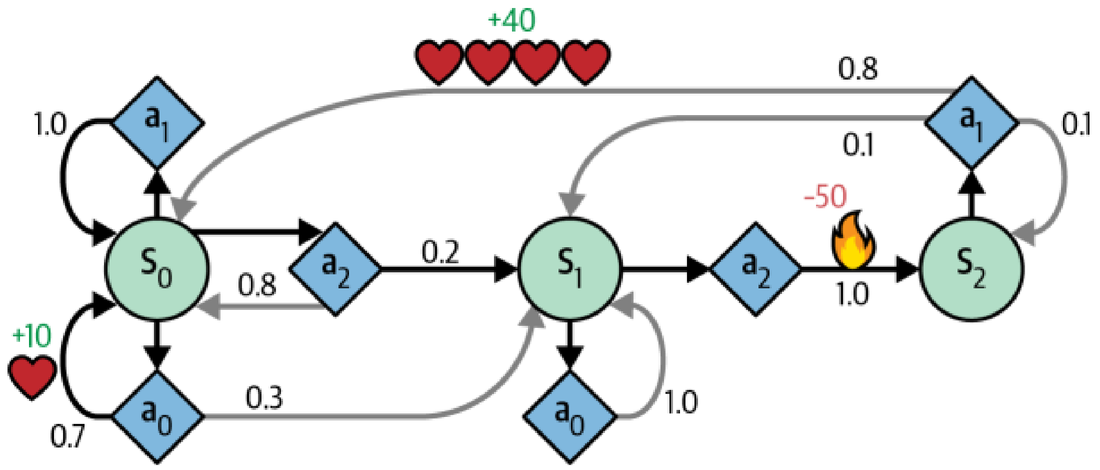
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

  - $T(s,a,s')$ is the transition probability from state s to state s', given that the agent chose action a.
  - $R(s,a,s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a.
  - γ is the discount factor.

Example

- Bellman optimality equation:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\,[R(s,a,s') + \gamma V^*(s')]$$



$$T(s_2, a_1, s_0) = 0.8 \quad R(s_2, a_1, s_0) = +40$$

Value Iteration Algorithm

- The Bellman optimality equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state:
  1. Initialize all the state value estimates to zero.
  2. Iteratively update the state values using the equation*:*

$$V_{k+1}(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

$V_k(s)$ is the estimated value of state s at the k-th iteration of the algorithm.

- Knowing the optimal state values can be useful to evaluate a policy, but it does not give us the optimal policy for the agent.

- Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-values* (quality values).

Q-Value Iteration Algorithm

- $Q^*(s,a)$ or the optimal Q-value of the state-action pair (s,a), is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.
- Q-value iteration algorithm:
  1. Initialize all the Q-value estimates to zero.
  2. Iteratively update the Q-values using the equation:

$$Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')\,[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')]$$

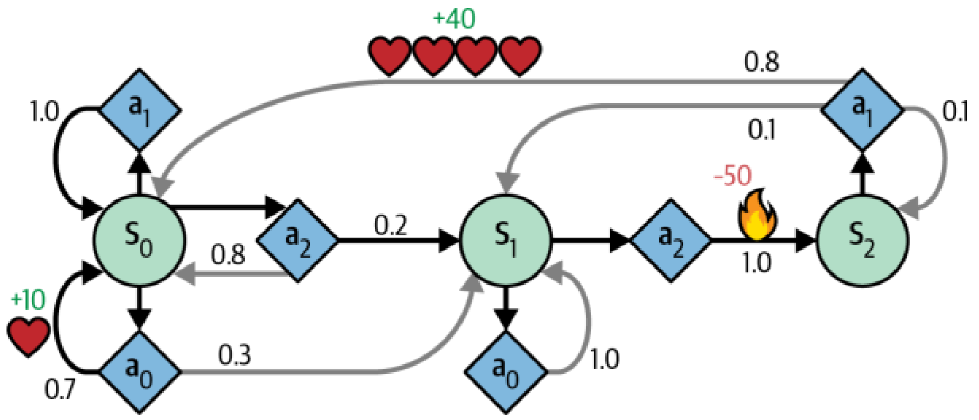
- Having the optimal Q-values, defining the optimal policy $\pi^*(s)$, is trivial:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a)$$

Example

- To know the transition probability of going from s_2 to s_0 after playing action a_1, we will look up `transition_probabilities[2][1][0]`
- To get the corresponding reward, we will look up `rewards[2][1][0]`.
- To get the list of possible actions in s_2, we will look up `possible_actions[2]`.

```
transition_probabilities = [  # shape=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]
]
rewards = [  # shape=[s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]
]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```



Running Q-Value Iteration Algorithm

- Initialize all the Q-values to zero (except for the impossible actions, for which we set the Q-values to $-\infty$):

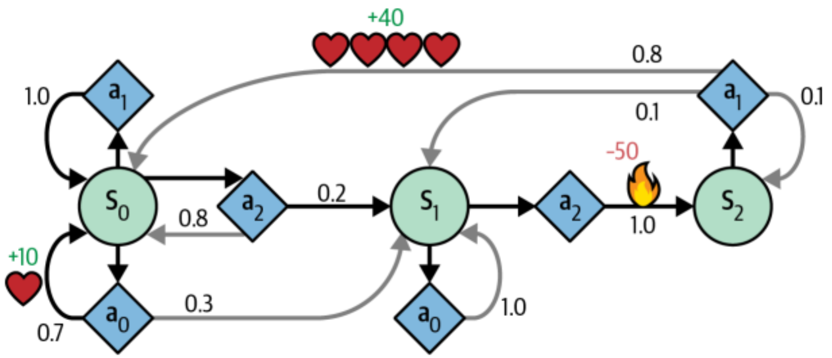
```python
Q_values = np.full((3, 3), -np.inf)  # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

- Run the Q-value iteration algorithm:

```python
gamma = 0.90  # the discount factor

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * Q_prev[sp].max())
                for sp in range(3)])
```

Example



- The resulting Q-values:

```
Q_values
```

```
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,        -inf, -4.87971488],
       [       -inf, 50.13365013,        -inf]])
```

- For each state, we can find the action that has the highest Q-value:

```
Q_values.argmax(axis=1)  # optimal action for each state
```

```
array([0, 0, 1])
```

Partial Knowledge of the MDP

- RL problems with discrete actions can often be modeled as MDPs, but the agent initially does not know :
  - the transition probabilities $T(s, a, s')$
  - the rewards $R(s, a, s')$
- The agent must experience each state and each transition at least once to know the rewards, and must experience them multiple times to have a good estimate of the transition probabilities.
- The *temporal difference (TD) learning* algorithm is similar to the Q-value iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP.

TD Learning Algorithm

- The agent initially knows only the possible states and actions.
- It uses an *exploration policy* (e.g. a purely random policy) to explore the MDP, and as it progresses, the TD learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed:

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma . V_k(s'))$$

or equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha . \delta_k(s,a,s')$$

with $\underbrace{\delta_k(s,r,s')}_{\text{TD Error}} = \underbrace{r + \gamma . V_k(s')}_{\text{TD Target}} - V_k(s)$