Report: Towers of Hanoi

Richard Polzin – Student Number : i6145946

Michael Zacharias – Student Number : i6146734

Introduction

This report summarizes the results of the Towers of Hanoi practical assignment for the course Foundation of Agents. It is part of the master Artificial Intelligence at Maastricht University.

The assignment was solving the Tower of Hanoi problem with three pins and two disks. It was only possible to move one disk at a time and there is a chance of making mistakes. This problem should be modeled as an MDP and solved using Q-Learning.

The report will describe the states and actions as well as the q-Value. The utility of each state for that policy will be noted and the convergence speed will be analyzed. Finally, the Q-Learning will be discussed.

States and Actions

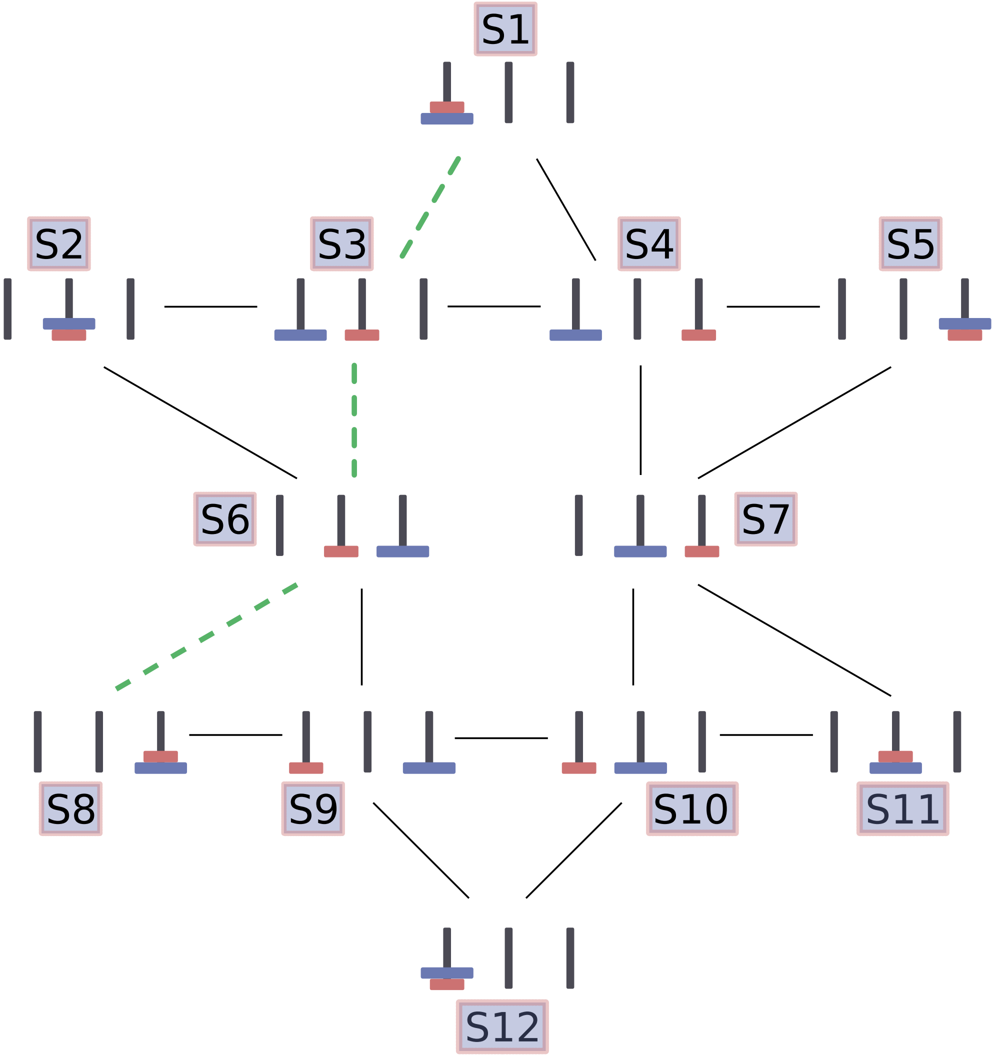

Illustration 1: Visualization of the State-Space

Illustration 1 shows the state-space. The starting state is S1. The fastest finishing moves are highlighted. Every line connecting a state to another corresponds to an action for those states. All the actions are bidirectional. The final state is state S8. Actions leading to S8 are not bidirectional. On every state the agent can decide to do nothing. This is achieved by introducing an implicit action leading from *s* to *s* at every state *s*. It yields a reward of *-1* if no bigger disk is on top of a smaller one. If the big disk is on top of the small one staying in that state adds a reward of *-10*. In the final state there is no punishment for staying in the state. Furthermore this is the only possible action in that state.

Optimal Policy

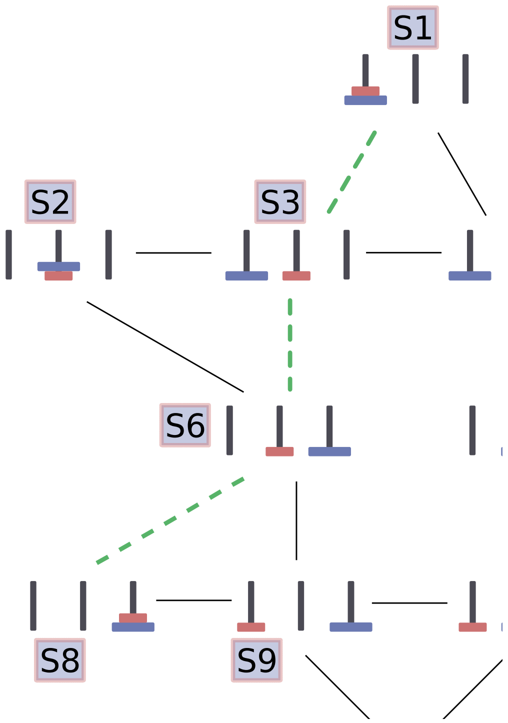

Illustration 2: Visualization of the fastest finishing moves

The fastest finishing moves are highlighted in illustration 2. First the small disk is moved to the center pin (S3). Then the big disk is moved to the rightmost pin (S6). Finally the small disk is positioned on top of the big disk on the right S(8). In conclusion the problem is solvable in three moves.

Value and Policy iteration both calculate utilities for every state. The best move is the one with the highest utility.

Table 1 shows the utilities that are calculated by the Policy iteration. Those are the utilities that are reached when the optimal policy is calculated.

Initial State *s*	**Action = Move to State *s'***	**Utility**	**Q-Value**
S1	S3	75.387	79.1
S2	S6	86.754	89.0
S3	S6	85.929	89.0
S4	S3	75.387	79.1
S5	S4	66.848	70.2
S6	S8	98.791	100
S7	S10	75.387	79.1
S8	S8	0	0
S9	S8	98.791	100
S10	S9	85.929	89.0
S11	S10	75.387	89.0

Table 1: Optimal Utilities and best Actions based on Policy Iteration

Illustration 3 visualizes the best actions for every possible state. The optimal policy for every state is indicated through the arrows. Analyzing the resulting actions shows that all the actions lead towards the goal state as expected.

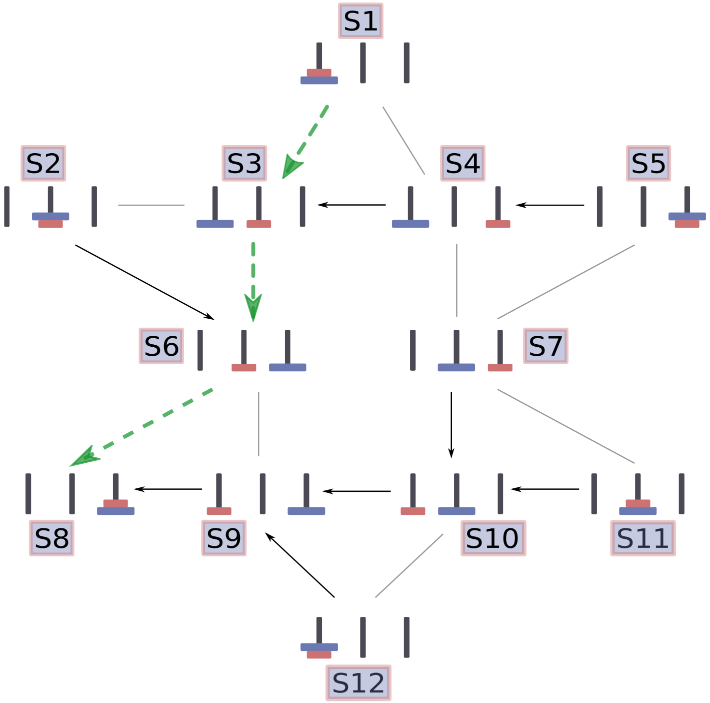

Illustration 3: Visualization of the State-Space and the Optimal Policy

Convergence Speed

Table 2 and Illustration 4 show how many iterations the Value iteration needed to reach a improvement smaller than epsilon.

Epsilon	Iterations
10	4
1	5
1×10^{-1}	6
1×10^{-5}	8
1×10^{-10}	10
1×10^{-15}	12

Table 2: Overview of different setups for the Value iteration

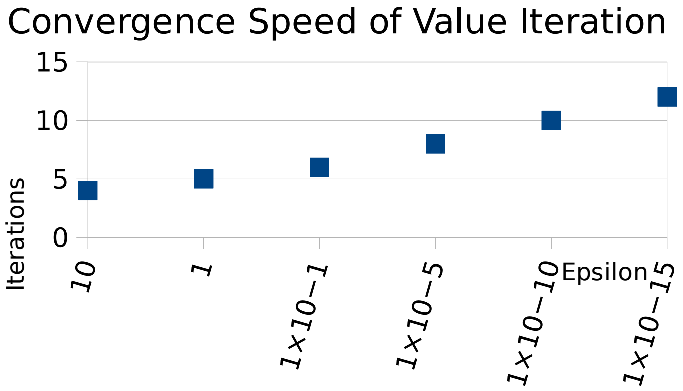

Illustration 4: Convergence Speed

Discussion

The main focus of this part is to elaborate the Q-Learning method used to solve the Tower of Hanoi problem. First of all, Q-Learning will be described and afterwards, the effort in implementation and general outlines will be presented.

Q-Learning is a method that can be used when the transition function is unknown. The problem in this case is, that policy and value iteration cannot be executed in this case, since both of them rely on the information of the transition function. Therefore, Q-Learning has to be used in order to identify the optimal policy.

In order to improve the policy, the agent explores the possible actions on every state. A random starting state is picked, as well as a random action to execute. The agent observer the result of the action and continues it's exploration from the resulting state. As soon as the final state is reached the process is started again. A learning rate can be used to determine whether the agent prioritizes exploration or solving of the problem. As we want to solely improve the policy a learning rate of 1 – always explore – is used.

From a mathematical point of perspective a q-value is stored for every action that is executable at a state. When this action is executed, it's q-value is updated. The new q-value is calculated in the following way (source: Wikipedia) :

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

As the learning rate is fixed to 1 the formula is reduced to a sum of the reward for reaching the state and the estimate of optimal future value times the discount factor. The estimated optimal future q-value is simply the highest obtainable q-value from the state we reach after executing the action.

In comparison to value/policy iteration the advantag of Q-Learning is the ability to deal with new situations. The agent does not require a complete model of all states, tranisitions and utilities but can handle unexplored states. While the process is slower than value/policy iteration it can be used to work with setups where neither value nor policy iteration are applicable.

A noteworthy aspect of Q-Learning is that the generated q-values differ from the utilities generated with the other approaches. This is because of a different calculation approach. Reaching a final state in value/policy iteration cannot generate a reward of 100. The failure rate of executing the action leading there limit the highest achievable reward to 100*0.9 + 0.1*X. Due to another calculation approach in Q-Learning the highest possible q-value is the same as the highest possible reward of reaching the state yielding the highest reward.

Code

```
"""
Basic Idea : The Tower of Hanoi problem will be solved. For this three pins (1,2 and 3) and two disks (A and B) are used.
Disk A is larger than B. The goal is to move the two disks to pin 3 such that the larger disk A is at the bottom and the
smaller disk B is at the top.
Markov Decision Process will be used to solve the problem. For this we define different rewards (first draft):
    - Reaching the goal : 100
    - Placing a larger disk on top of a smaller disk : - 10 (Penalty)
    - Every other step : -1 (force the agent to solve the problem with minimal steps)
The rules of the game are:
    - Only on disk may be moved at a time
    - Only the most upper disk from one of the rods can be moved
   [- It can only be put on another rod, if the rod is empty or a larger disk is on it]
Addition:
    The agent can make mistakes. When moving a disk from i to j, the agent may actually put the disk on pin k where
    k != i and k != j. The probability of this mistake is 10 %.
MDP:
    The general idea of a MDP is, that you model a problem using states and actions. Actions can move the agent from
    one state to another. With every action a probability is associated that the action actually yields the desired result.
    Furthermore a reward is defined for every transition from one state to another. To work with possible future state a
    discount factor is implemented, that states how important "future rewards" are compared to current rewards.
    The discount factor of future rewards is 0.9
    The problem will be solved using both Value Iteration and Policy Iteration.
Implementation:
    Take care, there are a few hickups:
        1. Actions described as an initial and final state
"""
import sys
import logging
import random
import numpy as np
from copy import deepcopy
import itertools
from collections import namedtuple
logging.basicConfig(level=logging.INFO)
Action = namedtuple('Action', 'InitialState FinalState')
Transition = namedtuple('Transition', 'Action Probability')
np.set_printoptions(edgeitems=3,infstr='inf', linewidth=300, nanstr='nan', precision=8, suppress=False, threshold=1000, formatter=None)
def gen_actions(s, d=None):
    """Generates all possible actions from a base state.
    :param s The State
    :param d The moved disk. If not None only action that move disk d are returned."""
    actions = [Action(s, s)]
    if s[-1] == [2, 1]:
        return actions
    for idx_out, pin in enumerate(s):
        for idx, move_to in enumerate(s):
            if pin != move_to and len(pin) != 0 and (d is None or pin[-1] == d):
                new_state = deepcopy(s)
```

```python
                del new_state[idx_out][-1]
                new_state[idx].append(pin[-1])
                actions.append(Action(s, new_state))
    return actions
def get_disk(a):
    """Return the disk that is moved from in the transition from state to action.
    :param a The Action"""
    for i in range(len(a.InitialState)):
        if a.InitialState[i] != a.FinalState[i]:
            return set(a.InitialState[i]).symmetric_difference(set(a.FinalState[i])).pop()
def t(a):
    """Returns the possible transitions trying to execute an action on a state.
    The desired state s' has a 90% success rate. Every different state has an equal probability.
    :param a The Action that should be executed"""
    if a.InitialState == a.FinalState:
        return [Transition(a, 1.0)]
    disk = get_disk(a)
    transitions = []
    actions = gen_actions(a.InitialState, disk)
    for g_a in actions:
        if g_a.FinalState == a.FinalState:
            p = 0.9
        elif g_a.FinalState == a.InitialState:
            p = 0.0
        else:
            p = 0.1 / (len(actions) - 2)
        transitions.append(Transition(g_a, p))
    return transitions
def r(a, s=None):
    """Returns the reward for reaching the FinalState of Action a.
    :param a The Action
    :param s The State. If not None the sum of all rewards and their probabilities is returned.
    """
    if s is not None:
        transitions = t(a)
        return sum([tr.Probability * r(tr.Action) for tr in transitions])
    else:
        for pin in a.FinalState:
            if not all(pin[i] >= pin[i + 1] for i in range(len(pin) - 1)):
                return -10
        else:
            if a.FinalState[-1] == [2, 1]:  # bad hardcoded win condition
                return 100 if not a.InitialState == a.FinalState else 0
            else:
                return -1
best_utility = []
def u(s, update=None):
    """Calculates or updates the current utility for a state.
    If no utility is present it is initialized to 0.
    :param s The State
    :param update If not None the utility for s is updated to update."""
    global best_utility
    for idx, us in enumerate(best_utility):
        if s in us:
            us[1] = update if update is not None else us[1]
            return us[1]
    else:
        initial_utility = 0
        best_utility.append([s, initial_utility])
        return initial_utility
def unique(iterable):
    """Used to adjust the itertool permutations method to our needs.
    http://stackoverflow.com/questions/6534430/why-does-pythons-itertools-permutations-contain-duplicates-
when-the-original
    :param iterable The iterable."""
    seen = set()
    for i in iterable:
        if str(i) in seen:
            continue
        seen.add(str(i))
        yield i
def id(s):
    """Returns the id of a State.
    :param s The State."""
    for i, os in enumerate(states):
        if os == s:
            return i
    else:
        raise Exception("State not in States.")
iterations = 0
def value_iteration(e):
    """Executes a value iteration.
    :param e The epsilon value. Search is halted when the difference is smaller than e."""
    while True:
        global iterations
        iterations += 1
        states_delta, states_best = [], []
        for s in states:
            ua_mapping = {}
            for a in gen_actions(s):
                reward = r(a, s)
                utility = reward + 0.9 * sum([tr.Probability * u(tr.Action.FinalState) for tr in t(a)])
                ua_mapping[utility] = a
            best_u = max(ua_mapping, key=float)
            best_a = ua_mapping[best_u]
            states_delta.append(abs(u(s) - best_u))
            states_best.append((best_a, best_u))
            u(s, best_u)
        if all([d < e for d in states_delta]):
            logging.info("Finished after {} iterations!".format(iterations))
            for b in states_best:
                logging.info("For State {} moving to State {} is best, with utility
{}".format(b[0].InitialState,

b[0].FinalState, b[1]))
            iterations = 0
            return
class Policy:
    def __init__(self):
        self.policy = [[s, gen_actions(s)[-1]] for s in states]
    def update(self, s, a):
        self.get_policy(s)[1] = a
    def get_action(self, s):
        return self.get_policy(s)[1]
    def get_policy(self, s):
        return [p for p in self.policy if p[0] == s][0]
def policy_iteration(e, p):
    """Executes a policy iteration to solve the MDP.
    :param e The epsilon value. Search is halted when the difference is smaller than e.
    :param p The current policy. Iteratively improved."""
    global iterations
    iterations += 1
    leq_a, leq_b = [], []
    for s in states:
        ids, part_leq_a = {}, []
        a = p.get_action(s)
        reward = r(a, s)
        for tr in t(a):
            ids[id(tr.Action.FinalState)] = tr.Probability
        for i in range(len(states)):
            if i in ids:
                if i == len(leq_b):
                    part_leq_a.append(1.0)
                elif i == id(a.FinalState):
                    part_leq_a.append((-0.9 * ids[i]))
                else:
                    part_leq_a.append(-0.9 * ids[i])
            else:
                part_leq_a.append(0)
        leq_a.append(part_leq_a)
        leq_b.append(reward)
    utility = np.linalg.solve(np.array(leq_a), np.array(leq_b))
    updated_policy = False
    for s in states:
        state_id = id(s)
        policy_action = p.get_action(s)
        other_actions = [a for a in gen_actions(s) if a != policy_action]
        for a in other_actions:
            other_utility = r(a, s) + 0.9 * sum(
                [tr.Probability * utility[id(tr.Action.FinalState)] for tr in t(a)])
            if other_utility > utility[state_id]:
                utility[state_id] = other_utility
                p.update(s, a)
                updated_policy = True
    if updated_policy:
        policy_iteration(e, p)
    else:
        for p in p.policy:
            logging.info("For State {} moving to State {} is best, with utility {}".format(p[0],
p[1].FinalState,

utility[id(

p[0])]))
        logging.info("Finished after {} iterations !".format(iterations))
        iterations = 0
tries = {}
def qlearning():
    qvalues = qlearningiter()
    for i in range(100):
        if i % 10 == 0:
            logging.debug("{}% done".format(i))
        qvalues = qlearningiter(qvalues)
    max = np.max(qvalues, axis=1)
    maxid = qvalues.argmax(axis=1)
    maxid[2] = 2  # ugly hack to fix value for absorbing state. row is (0, 0, .... , 0) so max returns first
0
    for i, s in enumerate(states):
        logging.info("For State {} moving to State {} is best, with utility {}".format(s, states[maxid[i]], 
max[i]))
def qlearningiter(qvalues = None, sid = None):
    qvalues = np.zeros((len(states),len(states))) if qvalues is None else qvalues
    sid = random.randint(0,len(states)-1) if sid is None else sid
    actions = gen_actions(states[sid])
    if len(actions) == 1:  # absorbing state
        return qvalues
    a = random.choice(actions)
    afst = states[np.random.choice([id(tr.Action.FinalState) for tr in t(a)], p=[tr.Probability for tr in 
t(a)])]
    a = Action(a.InitialState, afst)
    maxq = max([qvalues[id(act.InitialState),id(act.FinalState)] for act in gen_actions(a.FinalState)])
    q = r(a) + ( 0.9 * maxq )
    qvalues[sid, id(a.FinalState)] = q
    return  qlearningiter(qvalues, id(a.FinalState))
states = [i for i in unique(itertools.permutations([[2, 1], [], []]))] + \
            [i for i in unique(itertools.permutations([[2], [1], []]))] + \
            [i for i in unique(itertools.permutations([[1, 2], [], []]))]
epsilon = 0.00001
logging.info("Starting value iteration with an epsilon of {}".format(epsilon))
value_iteration(epsilon)
logging.info("Starting policy iteration")
policy_iteration(epsilon, Policy())
logging.info("Starting qlearning")
qlearning()
```