Report: Towers of Hanoi

Richard Polzin – Student Number : i6145946

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 Value iteration and Policy iteration.

The report will describe the states and actions as well as the optimal policy. The utility of each state for that policy will be noted and the convergence speed will be analyzed. Finally the differences between Value Iteration and Policy Iteration will be discussed.

States and Actions

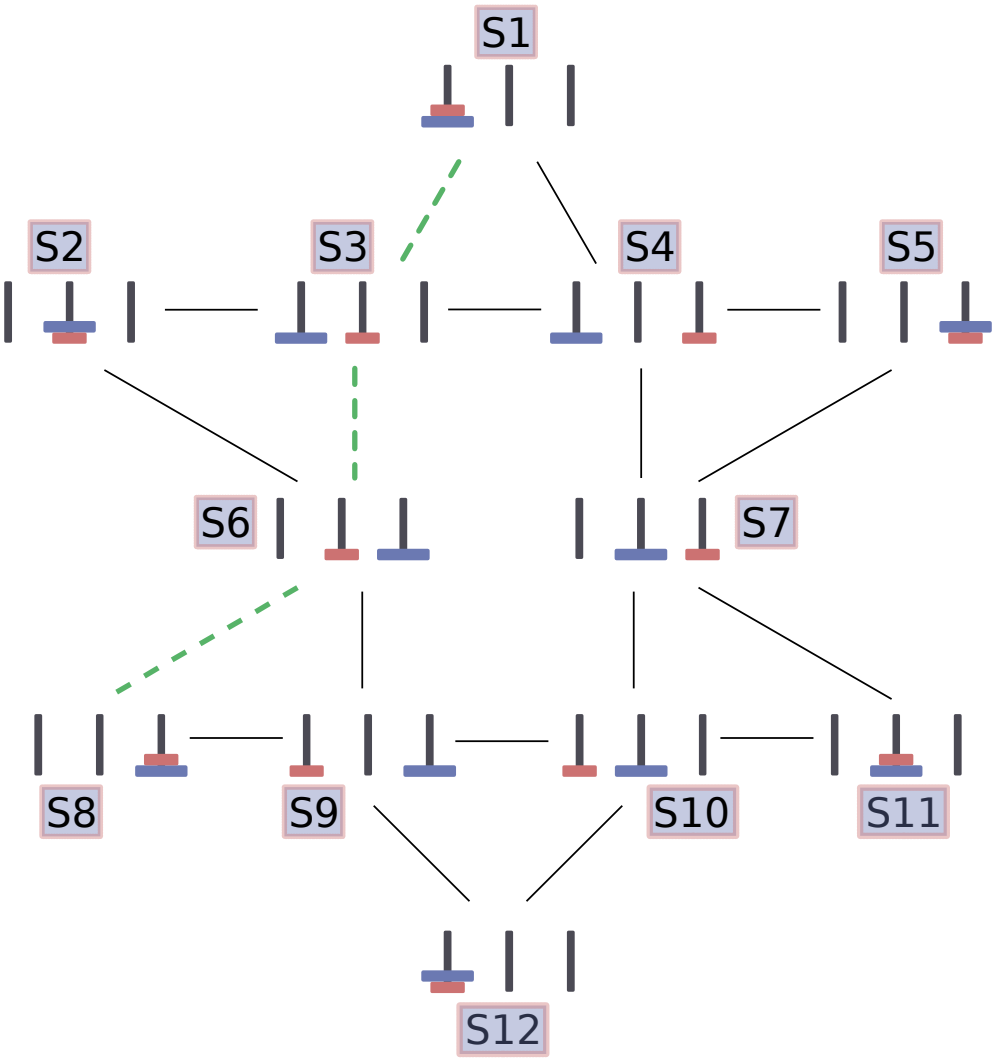

Illustration 1: Visualization of the State-Space

Illustration 1 shows the state-space. The starting state is S1. The principal variation is 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

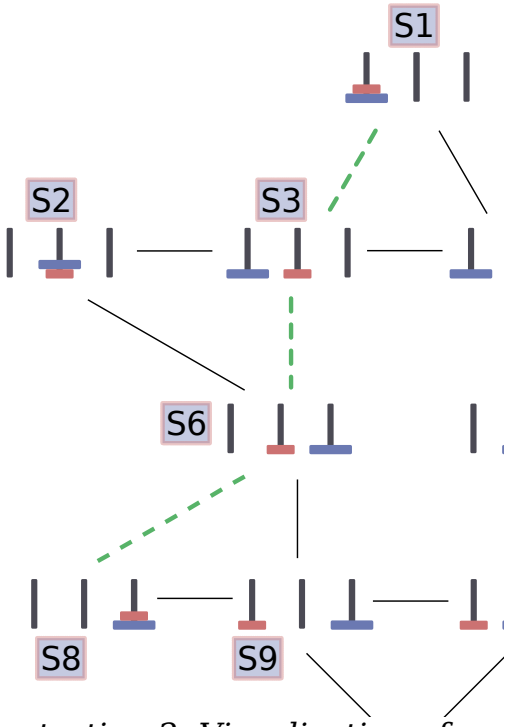

Illustration 2: Visualization of the Principal Variation

The optimal policy is highlighted in the state space as seen 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 utilites 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**
S1	S3	75.387
S2	S6	86.754
S3	S6	85.929
S4	S3	75.387
S5	S4	66.848
S6	S8	98.791
S7	S10	75.387
S8	S8	0
S9	S8	98.791
S10	S9	85.929
S11	S10	75.387

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

Illustration 3 visualizes the best actions for every possible state. The principle variation is highlighted again. Analyzing the resulting actions shows that all the actions lead towards at the goal state as expected.

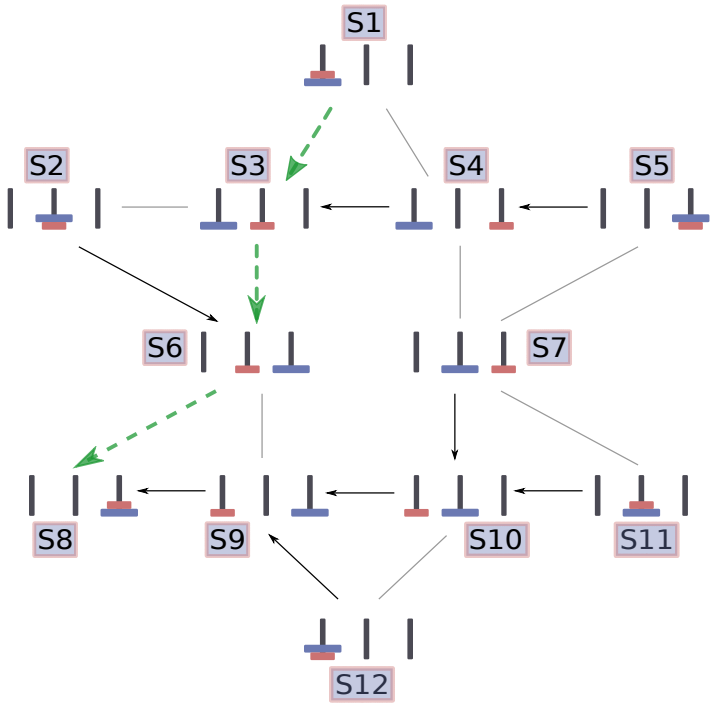

Illustration 3:
Visualization of the State-Space and the best Actions

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

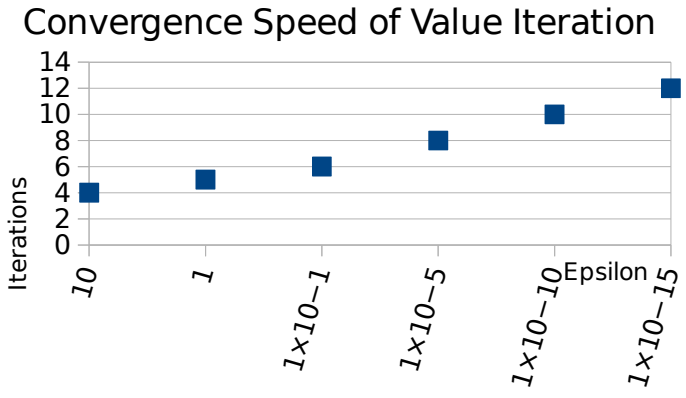

Illustration 4: Convergence Speed

Discussion

The main focus of this part is to elaborate on the difference between Value and Policy iteration. First of all Value and Policy iteration will be described on their own. The effort in implementation, the speed as well as the general outlines will be presented for each strategy. In the next part the advertised comparison will be executed.

Value iteration calculates the utility for every action in every state and updates it in every iteration. The action with the highest utility is assumed to be the optimal action. When the utility does not update more than a certain threshold the iteration is stopped. Implementing Value iteration is rather simple. The core of the implementation is a twofold loop that iterates on the states and every actions that can be executed on this state. In addition to that the calculated utilities have to be tracked and updated when a better utility becomes available. The time saved implementing Value iteration is lost again when it comes to performance. First of all with Value iteration only an approximate solution can be generated. Secondly the iteration speed can be pretty slow. More details will follow in the comparison.

Policy Iteration improves a policy in every iteration. The starting policy defines some action for every state. Utilities are calculated and compared to the utilities of the actions the policy did not choose. If an action is better than the one chosen by the policy, the policy is updated. When the policy does not change anymore the optimal policy is found. Implementing Policy iteration is more complex than value iteration. First utilities for the policies actions have to be calculated and the resulting linear equation has to be solved. Then the utilities for the other actions have to be calculated and compared to those chosen by the policy. Policy iteration always generates the guaranteed optimal policy, as it improves a policy until all the utilities are at their best. On the other hand solving a linear equation can become pretty expensive.

Comparing Value and Policy iteration the implementation effort differs noticeable but not significant. On the one hand Policy iteration generates the exact solution at the drawback of solving linear equations. Those can become pretty expensive for large state-spaces. On the other hand Value iteration generates an approximate solution with less computational effort, but at the risk of getting a policy that is not optimal.

In Conclusion Policy iteration should be used whenever sufficient. Policy iteration provides the exact solution which is favorable in most cases. Only if Policy iteration is too expensive, or optimality is not required Value iteration should be used.

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 logging
import numpy as np
from copy import deepcopy
import itertools
from collections import namedtuple

logging.basicConfig(level=logging.DEBUG)
Action = namedtuple('Action', 'InitialState FinalState')
Transition = namedtuple('Transition', 'Action Probability')


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)
                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 get_id_of_state(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.debug("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[get_id_of_state(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 == get_id_of_state(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 = get_id_of_state(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[get_id_of_state(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.debug("For State {} moving to State {} is best, with utility
{}".format(p[0], p[1].FinalState,

utility[get_id_of_state(

p[0])]))
        logging.info("Finished after {} iterations !".format(iterations))
        iterations = 0


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())
```