Practical_3_Stochastic Dynamic Programming

October 10, 2023

1 Practical 3: Stochastic Dynamic Programming

Author: CAMERON STROUD

Student Number: n11552123

1.0.1 Learning Outcomes:

- Markov Decision Process
- Stochastic Dynamic Programming

We will require the following library for this practical (Import all necessary libraries before running the code):

[ ]:
```
import numpy as np
```

1.1 Part A: Stochastic Shortest Path

Recall the shortest path problem in Practical 2.

Tom, who resides in City "A", is planning a journey towards City "H". Given his limited funds, he has devised a strategic plan to spend each night during his expedition at the abode of a friend. Tom has friends in cities "B", "C", "D", "E", "F", and "G".

Tom is mindful of optimizing his energy expenditure and he is aware of the limited distances he can cover each day. On the first day of travel, he can comfortably reach City "B", "C", or "D". On the second day, he can reach City "E", "F", or "G". Ultimately, Tom can reach his destination, City "H", on the third day.

Particularly, in this practical, we consider a stochastic scenario. The energy consumed during travel is dependent on random factors, including weather, traffic, etc. We can model this randomness with a probability distribution. For simplicity, we will consider a finite and discrete distribution with 3 possible outcomes. To conserve energy and navigate his journey efficiently, Tom must strategically decide where to spend each night along the route. It's imperative for him to consider the energy requirements between cities, which are outlined in the subsequent table. By skillfully selecting his overnight stops, Tom can ensure his expedition is, in average, both cost-effective and successful.

Cities	B	C	D
A	0.1 -> 120 0.2 -> 240 0.7 -> 390	0.3 -> 120 0.2 -> 430 0.5 -> 320	0.6 -> 250 0.1 -> 140 0.3 -> 220

Cities	E	F	G
B	0.4 -> 350 0.1 -> 630 0.5 -> 700	0.2 -> 140 0.2 -> 900 0.6 -> 120	0.8 -> 400 0.1 -> 200 0.1 -> 300
C	0.2 -> 150 0.6 -> 500 0.2 -> 700	0.2 -> 540 0.2 -> 490 0.6 -> 330	0.3 -> 840 0.1 -> 120 0.6 -> 430
D	0.3 -> 150 0.4 -> 130 0.3 -> 570	0.2 -> 600 0.5 -> 900 0.3 -> 120	0.2 -> 420 0.1 -> 320 0.7 -> 930

Cities	H
E	0.1 -> 450 0.4 -> 730 0.5 -> 940
F	0.2 -> 190 0.5 -> 380 0.3 -> 740
G	0.3 -> 550 0.6 -> 610 0.1 -> 720

The left-hand side of the tables indicate the departure cities, while the top denotes the arrival cities. For example, the "(0.1, 0.2, 0.7),(120, 240, 390)" in first line represents that, when Tom drives from City "A" to "B", it will consumes energy 120 with probability 0.1, and 240 with probability 0.2, and 390 with probability 0.7. Consider the following questions:

1.1.1 Q1

By inspection of the costs, intuit the optimal path.

F -> H seems to have a good spread of costs, with 50% chance of 380, and a 20% chance of 190, which seems better than the E and G spreads. B -> F has a high likelihood of a very small cost, 80%, with 60% chance of 120 and 20% chance of 140, however, has a small chance of a very large cost of 900. This is better than D but C is more consistent, mitigating high cost. With the high chance of very small cost, B seems to be the better choice, although considering A -> B and A -> C, the decision is less clear. A -> B is likely to have a higher cost, but leads into a high chance of minimal cost; while A -> C may result in a higher cost, it is more consistently lower, and has a higher likelihood of minimal cost, but leads into mid-range cost with C -> F. I intuit that A -> B -> F -> H is the optimal path.

1.1.2 Q2

Complete the following code to implement the stochastic dynamic programming algorithm for this stochastic shortest path (SPP) problem.

[ ]:
```
# Define the nodes at each step. Here, the nodes are defined by a dictionary.
  ↪The keys in this dictionary "0~3" represent the
# stage, and the values "0~7" represent City "A"~"H", respectively.
nodes = {
    0: [0],
    1: [1,2,3],
    2: [4,5,6],
    3: [7],
}
```

```
# Define the actions and the corresponding costs between the nodes. The keys in this dictionary "0~7" represent City "A"~"H",
# and the values corresponding to each key represent the next city, the probability distribution and the energy cost, respectively.
graph = {
    0: [(1, [0.1, 0.2, 0.7], [120, 240, 390]), (2, [0.3, 0.2, 0.5], [120, 430, 320]), (3, [0.6, 0.1, 0.3], [250, 140, 220])],
    1: [(4, [0.4, 0.1, 0.5], [350, 630, 700]), (5, [0.2, 0.2, 0.6], [140, 900, 120]), (6, [0.8, 0.1, 0.1], [400, 200, 300])],
    2: [(4, [0.2, 0.6, 0.2], [150, 500, 700]), (5, [0.2, 0.2, 0.6], [540, 490, 330]), (6, [0.3, 0.1, 0.6], [840, 120, 430])],
    3: [(4, [0.3, 0.4, 0.3], [150, 130, 570]), (5, [0.2, 0.5, 0.3], [600, 900, 120]), (6, [0.2, 0.1, 0.7], [420, 320, 930])],
    4: [(7, [0.1, 0.4, 0.5], [450, 730, 940])],
    5: [(7, [0.2, 0.5, 0.3], [190, 380, 740])],
    6: [(7, [0.3, 0.6, 0.1], [550, 610, 720])],
    7: [],
}
```

[ ]:
```
num_stage = len(nodes)
num_nodes = len(graph)
value_function = np.zeros(num_nodes)
value_function[num_nodes-1] = 0
optimal_action = np.zeros(num_nodes)
optimal_action[num_nodes-1] = num_nodes-1

# Stochastic dynamical programming algorithm
for k in range(num_stage-2, -1, -1):
    for n in nodes[k]:
        values = []
        num_action = len(graph[n])
        for a in range(num_action):
            transition = graph[n][a][0]
            probabilities = graph[n][a][1]
            costs = graph[n][a][2]
            # Hint: compute the expected value for each action. "np.dot" is an option.
            ### START CODE HERE ###
            expectation = sum([prob * cost for prob, cost in zip(probabilities, costs)]) + value_function[transition]
            values.append(expectation)
            ###  END CODE HERE ###

        value_function[n] = np.min(values)
        optimal_action[n] = graph[n][np.argmin(values)][0]
```

```
cities = ["A", "B", "C", "D", "E", "F", "G", "H"]

optimal_path_index = nodes[0]  # Initialize the optimal path with the starting␣
 ↪point
optimal_path = ["A"]
for k in range(1, num_stage):
    action = optimal_action[int(optimal_path_index[-1])]
    optimal_path_index.append(int(action))
    optimal_path.append(cities[int(action)])

print('Optimal Cost:', round(value_function[0],2))
print('Optimal Path:', optimal_path)
```

```
Optimal Cost: 1063.0
Optimal Path: ['A', 'B', 'F', 'H']
```

1.1.3 Q3

Does the optimal path provided by the algorithm match your intuition?

Yes, the algorithm produces the same result as my inuition.

1.1.4 Q4

Does the optimal path match the result obtained in Practical 2? Explain the similarities/differences.

The optimal path matches the results of practical 2 exactly. This is because the expectation results in the same cost as provided in practical 2. Even though there are different costs provided, the probability of each realisation results in the same expectation value given in the previous practical.

1.1.5 Q5

Modify the probabilities (ensure probabilities sum to 1) or the energy values, and compute the new optimal path and corresponding optimal cost. Discuss the differences observed in comparison to the initial scenario. Consider how these changes in uncertainties impact the outcomes and results.

[ ]:
```
# Define the nodes at each step. Here, the nodes are defined by a dictionary.␣
 ↪The keys in this dictionary "0~3" represent the
# stage, and the values "0~7" represent City "A"~"H", respectively.
nodes = {
    0: [0],
    1: [1,2,3],
    2: [4,5,6],
    3: [7],
}
```

```
# Define the actions and the corresponding costs between the nodes. The keys in this dictionary "0~7" represent City "A"~"H",
# and the values corresponding to each key represent the next city, the probability distribution and the energy cost, respectively.
graph = {
    0: [(1, [0.1, 0.2, 0.7], [120, 240, 390]), (2, [0.7, 0.1, 0.3], [120, 430, 320]), (3, [0.6, 0.1, 0.3], [250, 140, 220])],
    1: [(4, [0.4, 0.1, 0.5], [350, 630, 700]), (5, [0.2, 0.2, 0.6], [140, 900, 120]), (6, [0.8, 0.1, 0.1], [400, 200, 300])],
    2: [(4, [0.2, 0.6, 0.2], [150, 500, 700]), (5, [0.1, 0.2, 0.7], [540, 490, 330]), (6, [0.3, 0.1, 0.6], [840, 120, 430])],
    3: [(4, [0.3, 0.4, 0.3], [150, 130, 570]), (5, [0.2, 0.5, 0.3], [600, 900, 120]), (6, [0.2, 0.1, 0.7], [420, 320, 930])],
    4: [(7, [0.1, 0.4, 0.5], [450, 730, 940])],
    5: [(7, [0.2, 0.5, 0.3], [190, 380, 740])],
    6: [(7, [0.3, 0.6, 0.1], [550, 610, 720])],
    7: [],
}
```

[ ]:
```
num_stage = len(nodes)
num_nodes = len(graph)
value_function = np.zeros(num_nodes)
value_function[num_nodes-1] = 0
optimal_action = np.zeros(num_nodes)
optimal_action[num_nodes-1] = num_nodes-1

# Stochastic dynamical programming algorithm
for k in range(num_stage-2, -1, -1):
    for n in nodes[k]:
        values = []
        num_action = len(graph[n])
        for a in range(num_action):
            transition = graph[n][a][0]
            probabilities = graph[n][a][1]
            costs = graph[n][a][2]
            # Hint: compute the expected value for each action. "np.dot" is an option.
            ### START CODE HERE ###
            expectation = sum([prob * cost for prob, cost in zip(probabilities, costs)]) + value_function[transition]
            values.append(expectation)
            ###  END CODE HERE ###

        value_function[n] = np.min(values)
        optimal_action[n] = graph[n][np.argmin(values)][0]
```

```
cities = ["A", "B", "C", "D", "E", "F", "G", "H"]

optimal_path_index = nodes[0]  # Initialize the optimal path with the starting␣
 ↪point
optimal_path = ["A"]
for k in range(1, num_stage):
    action = optimal_action[int(optimal_path_index[-1])]
    optimal_path_index.append(int(action))
    optimal_path.append(cities[int(action)])

print('Optimal Cost:', round(value_function[0],2))
print('Optimal Path:', optimal_path)
```

```
Optimal Cost: 1056.0
Optimal Path: ['A', 'C', 'F', 'H']
```

Changing the probabilities of A -> C and C -> F has changed the optimal path from A -> B -> F to A -> C -> F, without a change in costs. This means that inappropriate statistical distribution of costs can result in suboptimal trajectories, as the expectations are not consistent with reality

1.2 Part B: Stochastic Transition Problem

Consider a new SPP scenario. Tom embarks from City "A" and is presented with two possible directions: "East" and "West". Each direction leads to a fork in the road. The "East" direction offers paths to City "B" and "C", while the "West" direction connects to City "D" and "E". Importantly, the possibility exists that one of these paths may be obstructed due to factors like a traffic accident or natural disaster. However, Tom can only ascertain which road is closed once he reaches the fork in the road. The graphical representation of this scenario is provided below:

The depicted graph indicates that the paths leading to City "B" and "C" could potentially be obstructed with probabilities of 0.4 and 0.6, respectively. Similarly, the paths to City "D" and "E" may experience closures with probabilities of 0.2 and 0.8, respectively. The primary goal is to determine the optimal action to take at each city in this scenario. The corresponding energy costs between the cities are provided below:

Cities	A	B	C	D	E	F	G	H
A	/	333	282	230	300	/	/	/
B	/	/	/	/	/	553	280	/
C	/	/	/	/	/	470	404	/
D	/	/	/	/	/	268	606	/
E	/	/	/	/	/	807	370	/
F	/	/	/	/	/	/	/	450
G	/	/	/	/	/	/	/	603

1.2.1 Q6

Complete the following code to implement stochastic dynamic programming algorithm for the above stochastic SPP.

[ ]:
```
# Define the nodes at each step. Here the nodes are defined by a dictionary.␣
 ↪The keys in this dictionary "0~3" represent the
# stage, and the values "0~7" represent City "A"~"H", respectively.
nodes = {
    0: [0],
    1: [1,2,3,4],
    2: [5,6],
    3: [7],
}

# Define the actions and the corresponding costs between the nodes. The keys in␣
 ↪this dictionary "0~7" represent City "A"~"H",
# and the values corresponding to each key represent the next city and the␣
 ↪energy cost between these two cities, respectively.
graph = {
    0: [([1,333], [2,282]), ([3,230], [4,300])],
    1: [(5,553), (6,280)],
    2: [(5,470), (6,404)],
    3: [(5,268), (6,606)],
    4: [(5,807), (6,370)],
    5: [(7,450)],
    6: [(7,603)],
    7: [],
}

# Define the transition probability matrix
trans_prob = np.array([[0.4,0.6], [0.2,0.8]])
```

[ ]:
```
num_stage = len(nodes)  # The number of stages
num_nodes = len(graph)  # The number of nodes
value_function = np.zeros(num_nodes)  # Initialize the value function for each␣
 ↪node
value_function[num_nodes-1] = 0
optimal_action = []
optimal_path_index = nodes[0]  # Initialize the optimal path with the starting␣
 ↪point

cities = ["A", "B", "C", "D", "E", "F", "G", "H"]  # The city nodes
directions = ["East", "West"]

# Implement deterministic dynamical programming algorithm
for k in range(num_stage-2, -1, -1):
    for n in nodes[k]:
        values = []
        num_action = len(graph[n])

        if n==0:
```

```
            for a in range(num_action):

                # Hint: compute the value for each action in City "A", and␣
 ↪select the action with minimum value
                # "optimal_action.insert(0, value)" is to place the value at␣
 ↪the forefront of the "optimal_action" list
                ### START CODE HERE ###
                transition = graph[n][a]
                probabilities = trans_prob[a]
                costs = [action[1] + value_function[n] for action in transition]

                expectation = sum([prob * cost for prob, cost in␣
 ↪zip(probabilities, costs)])
                values.append(expectation)

            value_function[n] = np.min(values)
            optimal_action.insert(0, directions[np.argmin(values)])

        else:
            for a in range(num_action):

                # Hint: compute the value for each action in other cities, and␣
 ↪select the action with minimum value
                ### START CODE HERE ###
                node = graph[n][a][0]
                cost = graph[n][a][1]  # Extract the cost from the action
                value = cost + value_function[node]
                values.append(value)

            value_function[n] = min(values)  # Update the value function for␣
 ↪the current node

            ### END CODE HERE ###

# Print the results
print('Optimal Cost:', value_function[0])
print('Optimal Action:', optimal_action)
```

```
Optimal Cost: 286.0
Optimal Action: ['West']
```

1.3 Part C: Parking Problem

Let's delve into the parking problem. The parking problem refers to a scenario where you want to optimally park a car in a parking lot while minimizing a cost. Consier the following example: A driver is looking for a park on a street with $N-1$ car parks. The driver can stop in any car park with cost $c(k)$ and starts looking at car park $k=0$. If the driver has not stopped by car park

$k = N - 1$ then the driver must park at the expensive multi-story car park (terminal state) with cost C. Each car park k has an independent random chance of being free with probability $p(k)$. We will consider the following cost functions and probabilities: - (a) $c(k) = N - k,\ p(k) = 0.01$ - (b) $c(k) = -k^2,\ p(k) = 0.01$ - (c) $c(k) = k - N,\ p(k) = 0.01$ - (d) $c(k) = k,\ p(k) = \min(1/k, 0.001)$

1.3.1 Q7

Consider scenario (a), what do you think will occur? Considering, an average cost, where is the best point to stop?

For the cost in (a), it would seem that there is a linear decrease in cost as the car explores, resulting in prioritising car parks closer to end, and, due to the low probability of finding a car park, will likely reach the terminal state.

1.3.2 Q8

Discuss the purpose of a terminal state (in the context of a finite horizon).

Terminal states are important in order to bias the result of the optimisation in a particular way. In the case of the car park, we want to find a car park, even if its far away, but we still want to get as close as possible. With a finite horizon, incorporating the terminal state allows us to prioritise the latter until the terminal state is 'within view', then penalise as we get closer.

1.3.3 Q9

Write this problem as a Markov Decision Process. (Hint: Identify states and actions)

S = {0, 1, 2 … N-1, T} where T = Terminal State
A = {0 (search), 1 (park)}

P_park =
[1, 0, … 0]
[0, 1, … 0]
[:, …]
[0, … , 1]

1.3.4 Q10

Complete the following code to implement stochastic dynamic programming algorithm for the parking problem.

[ ]:
```
#Define the stochastic dynamic programming function
def stochastic_dynamic_programming(parks_number, search_cost, final_cost,␣
 ↪free_probability):
    # Create a value function table with the size of the parks_number
    value_function = np.zeros(parks_number)
    value_function[parks_number-1] = final_cost

    # Create a policy table to store the optimal actions
    policy = np.zeros(parks_number)
```

```
    # Iterate over each park, starting from the last one
    for k in range(parks_number - 2, -1, -1):
        parking_cost = - (k**2)

        # Hint: calculate the value of parking at the current park and␣
 ↪searching for another park
        ### START CODE HERE ###
        park_value = parking_cost + (1 - free_probability) * value_function[k +␣
 ↪1]
        search_value = search_cost + free_probability * value_function[k + 1]
        ### END CODE HERE ###


        # Hint: update the minimum cost and the best action if necessary
        ### START CODE HERE ###
        if park_value < search_value:
            value_function[k] = park_value
            policy[k] = 1  # 1 corresponds to parking
        else:
            value_function[k] = search_value
            policy[k] = 0  # 0 corresponds to searching
        ### END CODE HERE ###

    return value_function, policy
```

1.3.5 Q11

Utilising your stochastic dynamic programming function, compute the cost of each state for the above scenarios.

[ ]:
```
parks_number = 10
search_cost = 1
final_cost = 10
free_probability = 0.01

value_function, optimal_policy = stochastic_dynamic_programming(parks_number,␣
 ↪search_cost, final_cost, free_probability)
print("Value function:", value_function)
print("Optimal policy:", optimal_policy)
```

```
Value function: [-182.2723538  -184.11348869 -184.96311989 -182.79103019
-175.54649514
 -161.1580759  -137.53341    -102.559       -54.1          10.        ]
Optimal policy: [1. 1. 1. 1. 1. 1. 1. 1. 1. 0.]
```

1.3.6 Q12

What is the optimal policy for each scenario?

The optimal policy is to continue searching, in every scenario

1.3.7 Q13

Does the computed optimal control policy align with your intuitive expectations?

It does, as the cost penalises early parking over later parking. The search cost also only depends on the future cost, and not the current state, resulting in a relatively low cost to search when compared with parking cost.

1.3.8 Q14

Experiment with altering the cost functions and probabilities, and observe the resultant variations in the optimal policy. Discuss how these parameter adjustments influence the determination of the optimal policy.

Due to the way the parking and search costs are determined, the terminal state cost has no impact on the resulting policy, only scaling the resulting costs. Changing the scenario to (b) results in a preference for parking early, but after a certain point it stops attempting to park and searches, the inverse of the desired behaviour.