International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V72I7P105

Volume 72 Issue 7, 32-52, July 2024

Original Article

Optimizing Payment Approvals: Dynamic Programming Approach

Manasa Gudimella[1], Aditya Gudimella[2]

[1]*Applied Data Scientist, Cumming, Georgia, United States.*
[2]*Senior Artificial Intelligence Engineer, R&D, Danville, California, United States.*

[1]*Corresponding Author : Manasa.gudimella@gmail.com*

Received: 11 May 2024 Revised: 22 June 2024 Accepted: 12 July 2024 Published: 30 July 2024

Abstract *- This paper introduces a smart payment system designed to optimize the selection of payment service providers for each transaction, aiming to increase payment approval rates using dynamic programming. This solution is applicable to any business that processes payments, as an increase in overall approval rates can enhance cash flow and reduce payment-related costs. To ensure payment system reliability and avoid single points of failure, transactions are distributed among providers within specified thresholds, thereby balancing the traffic allocation. This factor is integrated into the optimization model. Through simulated data, the proposed solution demonstrates its effectiveness in increasing transaction approval rates by employing a smart optimization policy that selects actions in each state to maximize total rewards. The effectiveness of the presented approach is demonstrated by comparing different strategies; the results show that revising the traffic allocation daily can improve the overall reward by 8.1% for simulated data.*

Keywords *- Dynamic Programming, Payment Optimization, Smart Payment Routing.*

1. Introduction

Customer payments to merchants are typically routed through intermediaries known as Payment Service Providers (PSPs), who facilitate transactions on behalf of the merchants. Each transaction often presents a choice of multiple providers, and selecting the optimal provider for a given transaction based on its specific features is crucial. PSPs employ various techniques to ensure successful retrieval of funds from customer accounts, and they continuously upgrade their technology to improve transaction approval rates. The approval rate is the proportion of transactions approved by the payment system over the total number of transactions that were processed by the system. Therefore, it is in the merchant's best interest to continually monitor the approval rates of different providers and allocate each transaction to the most effective one to maximize the overall transaction approval rates. Even a modest 1% increase in approval rates, achieved through this ongoing optimization process, can significantly boost the merchant's cash flow. For instance, a business processing 100,000 transactions per day with an average transaction size of $50 would see an increase of $50,000 in daily cash flow with just a 1% improvement in approval rates. Additionally, providers charge merchants a fee for processing each transaction. By optimizing and increasing the approval rate, merchants can avoid the costs associated with multiple processing attempts for a single transaction in case of initial failures, leading to significantly lower transaction costs. This optimization is particularly beneficial for small to medium-sized merchants, for whom the reduction in costs and the increase in successful transactions can have a substantial financial impact.

2. Problem

This article addresses the decision-making problem faced by merchants in handling customer transactions. Each day, customers generate a large number of transactions while shopping with merchants. Each transaction is unique, identified by a specific transaction ID and characterized by various attributes such as card type (credit, debit, prepaid) and network (MasterCard, Discover, Visa, Amex). Crucially, each transaction has an associated amount that the customer owes the merchant, referred to as the transaction's worth. Merchants have the option to route each transaction through one of multiple payment service providers. For simplicity, this article assumes that there are three fixed providers labelled as 1, 2, and 3. An effective dynamic payments system should assign each transaction to one of these providers according to a pre-specified traffic allocation distribution to maximize the overall transaction approval rates. For example, if the traffic allocation distribution is $p = (0.7,0.2,0.1)$, then 70% of transactions are assigned to provider 1, 20% to provider 2, and 10% to provider 3 in a random manner. The performance of each provider is evaluated based on the proportion of transactions that are successfully approved.

The allocation of traffic to the providers is revised periodically, with the frequency of revision determined by the merchant's transaction volume. For instance, a merchant processing 100,000 transactions daily has sufficient data points to revise the traffic allocation distribution daily with statistical significance. Based on this cadence, the traffic allocation distribution can be updated. This paper provides methodologies to:

1. Ensure the traffic allocation distribution is enacted correctly, ensuring each transaction is randomly assigned to a provider as per the set distribution. This assumption is critical for the proposed payment optimization.
2. Develop a payment optimization model.

For validating the effectiveness of randomness in provider assignment for transactions and ensuring that the transaction distributions in real-time match the pre-specified distributions, statistical tests such as the chi-square goodness of fit and runs test can be employed. Developing an optimal strategy for payment provider assignment involves formulating a policy and objective. The objective is to increase the total number of approved transactions. The period for evaluating overall returns must be clearly defined, as it is a matter of policy. During this period, decisions are made periodically based on dynamic business scenarios, and appropriate actions are implemented. In this article, the period is taken as 250 days (considering weekdays over one year), and the policy is to allow decisions (actions) to change the assignment distribution every day so that the total rewards achieved each day over the *planning horizon* of 250 days is maximized.

The business scenario at the beginning of each epoch (each day) is known as the state of the system. The state of the system for the problem in question at the beginning of any epoch is defined by the relative performance of the providers compared to their performance in the previous epoch. The optimization problem considered in this article involves defining and comparing a set of actions and evaluating different strategies in terms of their impact on overall returns over the planning horizon. At the start of each epoch, depending on the state of the system, an action is taken, resulting in a reward at the end of the epoch. The problem is to maximize the total rewards over the planning horizon. The state of the system at the beginning of any epoch depends on the state and the action of the previous epoch. The system states transition mechanism in practice is governed by the stochastic behavior of the system. One way of handling the problem is to consider a deterministic model by replacing the stochastic mechanism with expected behavior. The task of determining the sequence of actions to maximize the total reward is known as a dynamic programming problem.

3. Related Work

We focussed our literature survey on the application of AI and ML in enhancing smart payment solutions to boost transaction approval rates. Bygari [6] combined static rules with supervised learning models, such as logistic regression and random forest classifiers, to forecast provider probabilities, resulting in a 4-6% increase in transaction success rates. The multi-armed bandit approach was demonstrated by Gefferie [7] to select the optimal provider in the e-commerce domain effectively. Dream 11 Engineering [8] conducted offline simulations of various bandit algorithms, including Epsilon Greedy and Upper-Confidence-Bound (UCB), fine-tuning hyperparameters without the risks of real-time experimentation and achieved a significant uplift in the transaction success rates (approximately 0.92% monthly uplift).

Further studies [9] and [10] highlight the necessity for smart payment systems capable of operating autonomously or with expert oversight, smartly analyzing elements such as transaction fees and success rates to choose the most efficient provider. These developments indicate a move towards autonomous, AI-powered solutions in global payment routing, focusing on improving user experience and operational efficiency. Lyft [11] implemented a reinforcement learning (RL) platform, utilizing contextual bandits to build out a multi-decision system that addresses similar optimization problems through online learning. In the realm of commerce and finance, multi-armed bandits have been applied to solve comparable optimization challenges for portfolio construction [1]. Contextual bandits in RL are becoming increasingly popular, with substantial theoretical development, including the Epoch-Greedy Algorithm by Langford [3] and the Thompson Sampling algorithm for stochastic multi-armed bandits by Agarwal [2].

Research [4] plays a critical role in evaluating computationally efficient and optimal contextual bandit methods, offering guidance for practitioners. It critiques the application of Upper Confidence Bounds (UCB) and Thompson Sampling in managing sparse, high-dimensional datasets due to their strong modelling assumptions and challenges in practical implementation. It also explores a confidence-based method incorporating LinUCB, an advancement of UCB. Another key study [5] devised a machine learning system specifically for contextual decision-making, optimizing operations in environments like MSN.com. While these RL-based systems operate as independent decision-makers, their adoption requires significant engineering investment and a steep learning curve.

3.1. Contributions

This paper introduces the application of dynamic programming for developing an intelligent payment system. The proposed approach is accompanied by the corresponding code. Simulations have demonstrated the effectiveness of this methodology. The novelty of the tool lies in its simplicity of modelling approach, straightforward code, and ease of implementation and integration into existing systems.

4. Data Collection and Preprocessing

Real-world payment data typically encompass a rich set of variables that allow for the exploration and identification of relationships among features. Examples of such features include transaction card type, network, time of day, day of the week, specific business characteristics, customer segmentation based on type, location, transaction amount, and more. Initial data analysis can begin with examining the proportion of transactions assigned to providers and their corresponding approval rates. This analysis can reveal trends in approval rates based on the allocation of transactions to different providers. By utilizing historical data and additional features, the relationship between these variables can be modelled effectively. A preliminary step involves calculating simple descriptive statistics, such as the mean and standard deviation of the approval rates, both overall and for each categorical variable. This can help uncover trends. For instance, this may uncover differences in provider performance across various geographies.

Examining approval rates on a weekly basis for each provider can reveal if changes in approval rates follow similar trends across providers (e.g., simultaneous increases or decreases). Additionally, plotting provider performance over time and by geography with respect to the number of transactions can indicate geography-dependent performance, which may be identified through clustering. Autocorrelation functions are valuable performance measures. For example, they can be used to determine if the number of transactions and approval rates remain relatively stable from day to day. If a provider had a certain approval rate on a particular day, it could be examined whether a similar approval rate is likely on the following day. These data analyses are essential in exploring strategies for optimization.

5. Methodology

5.1. Validation of Data Distribution and Independence Assumptions

Customer transactions to merchants vary depending on the size and nature of the merchant. When selecting a provider for a transaction, the requirement is that the provider should be randomly selected according to the pre-specified traffic allocation distribution. It is necessary to validate if these best engineering practices are followed based on the data collected from the payment system. Consider a situation where there are three providers labelled as 1, 2, and 3. Let N be the total number of transactions to be distributed to the three providers on any day according to a pre-specified distribution, say $p = (p_1, p_2, p_3)$. To assign the transactions randomly for i^{th} transaction, $i = 1, 2, \dots, N$, pick a random number x_i from $\{1, 2, 3\}$ so that the probability of $x_i = j$ is equal to $p_j, j = 1$ 2, 3. Denoting the distribution of x_i by F, , $x_1, x_2, \dots, x_N$, is a sequence of random variables from F. If the transactions are assigned independently, then the sequence $x_1, x_2, \dots, x_N$ is a sequence of $i.i.d$ (independent and identically distributed) random variables. To validate the random allocation hypothesis (i.e., allocation is made independently), a subsequence $x_{i+1}, x_{i+2}, \dots, x_{i+n}$ where i is any random number picked randomly from $1\ to\ N - n$, which can be used. For ease of notation, the subsequence $x_{i+1}, x_{i+2}, \dots, x_{i+n}$ shall be denoted by $x_1, x_2, \dots, x_n$. One way to validate the hypothesis is to plot the autocorrelation function (ACF) of the sequence $x_1, x_2, \dots, x_n$. An alternative approach for this validation problem using Markov Chains is proposed below. A sequence of random variables $\{u_k, k = 1,2, \dots\}$ is a Markov chain with one-step stationary transition probabilities if:

$Prob(u_{k+1} = j \mid u_k = j_k, u_{k-1} = j_{k-1}, \dots, u_1 = j_1) = Prob(u_2 = j \mid u_1 = j_k)$

holds for all k and all states $j, j_k, j_{k-1}, \dots, j_1$. The matrix $P = (p_{ij})$, where $(i, j) - th$ element $p_{ij} = Prob(u_2 = j \mid u_1 = i)$, and i and j are states that are called the one-step transition matrix of the Markov chain. For background on Markov chains, refer to [15] [16].

If the sample assignment sequence x1, x2, ... satisfies the hypothesis with the pre-specified distribution $p = (p_1, p_2, p_3)$, then it forms a Markov chain with state space $S = \{1,2,3\}$ and stationary transition probabilities. Since any subsequence of the Markov Chain is also a Markov Chain, $x_1, x_2, \dots, x_n$ is also a Markov Chain. In this case, each row of the one-step transition matrix is p. In this case, each row of the one-step transition matrix is p. This fact can be utilized to test the hypothesis of randomness and distribution fit in the assignment. Henceforth, this hypothesis will be referred to as H. In other words, H indicates that $x_1, x_2, \dots, x_n$ is an $i.i.d$ sequence from F. More precisely, the probabilities (or the corresponding frequencies) of the one-step transition matrix are estimated and compared with their expected values under H.

The two methods are compared using simulated data. Two sequences are simulated – one satisfying H, and the other violating H. Simulating a sequence satisfying H is straightforward, but simulating a sequence under violation of H requires a procedure. This can be done by simulating a sequence from a Markov chain with a one-step transition matrix having distinct rows.

For this problem, the three rows of the one-step transition matrix are taken as follows: the first row as q1 = (0.75, 0.10,0.15), the second row as q2 = (0.70, 0.20, 0.10), and the third row as q3 = (0.70, 0.15, 0.10). This setup ensures that the simulated sequence violates H. Python code for simulating the data is provided in the Appendix. Two sequences are simulated, one satisfying H and the other violating it. The autocorrelation functions (ACFs) of the two sequences are shown in Figure 10. In both cases, all the autocorrelations are within ±1.96 standard deviation limits, indicating no substantial evidence for suspecting non-independence.

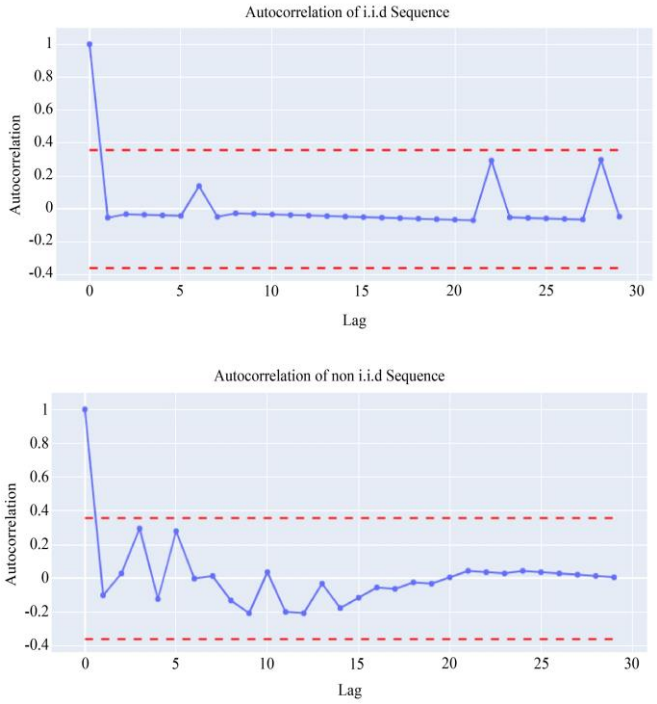


Fig. 1 ACFs of *i.i.d* and *non-i.i.d* sequences

To apply the Markov chain approach, the one-step transition frequencies are computed (see the Python code for obtaining the transition frequencies and one-step transition probabilities in Subsection 5.2). The top tables in Figure 11 present the transition frequencies. The corresponding expected frequencies are computed in the tables below them.

To test the match between observed and expected frequencies, the chi-square statistic is computed. The chi-square statistics $\sum(O_{ij} - E_{ij})^2 / E_{ij}$ values are calculated for each transition frequency and listed in Figure 2.

This follows an approximate chi-square (χ^2) distribution with 6 degrees of freedom with 95th percentile 13.27. From the results, the test rejects H for the non-$i.i.d$ sequence, while it fails to reject H for the $i.i.d$ sequence.

Table 1. Chi-square test for i.i.d and non-i.i.d sequences

Observed Transitions (*i.i.d*):			Observed Transitions (*non-i.i.d*):		
1006	284	124	1112	132	241
277	92	38	153	51	17
131	31	16	220	38	35
Expected Transitions (*i.i.d*):			Expected Transitions (*non-i.i.d*):		
989.8	282.8	141.4	1039.5	297	148.5
284.9	81.4	40.7	154.7	44.2	22.1
124.6	35.6	17.8	205.1	58.6	29.3
Chi-square Statistic (*i.i.d*): **5.30**			Chi-square Statistic (*non-i.i.d*): **166.02**		

5.2. Optimization of Total Rewards using Dynamic Programming Model

Payment optimization, which aims to increase the total number of approved transactions, is explored in this section using two different strategies adopting dynamic programming: the daily strategy and the fortnightly strategy. In the daily strategy, the traffic allocation distribution is revised each day, while in the fortnightly strategy, it is revised every two weeks. The state space, action space, rewards, and the chance mechanism that governs the movement of the system are identified. A simulation model will be used to generate data for comparison purposes. The dynamic programming model is employed to compare two strategies.

In a dynamic programming problem, the system begins in the state. s_0 at time 0, and an action a_0 is taken. This results in a reward r_0, and the system transitions to the state s_1 at time 1. Based on s_1, an action a_1 is taken, resulting in a reward r_1 and a transition to a state s_2 at time 2. This process continues until time T – 1. The final action a_{T-1} is taken at the beginning of time $T-1$, and a reward r_{T-1} is collected at the end of time $T-1$ or equivalently at the beginning of time T, where the system reaches state s_T. The total reward at this final state is given by: $R(s_T) = \sum_{t=0}^{T-1} r_t$. The objective is to choose the sequence of actions. $a_0, a_1, \ldots, a_{T-1}$ to maximize $R(s_T)$ over, all possible states s_T.

5.3. Modelling Transaction Allocation Problem

The transaction allocation problem is described as follows. Time units are defined as days, and the planning horizon consists of the working days in a year, represented by $H = \{0,1,2,\ldots,T\}$, where T may be considered to be 250 (number of weekdays in a year). The system is defined as follows: On day 0, the state of the system is s_0, with a probability distribution $p^0 = (p_1^0, p_2^0, p_3^0)$ and the corresponding approval rate vector q^{-1} are associated with it. The vectors p^0 and q^{-1} represent the previous allocation vector and the corresponding approval proportions, respectively. Let N_0 be the number of transactions to be distributed to the providers on day 0. On any given day, the system can be in one of 8 possible states derived from the performances of the providers. Let $q_d = (q_d^1, q_d^2, q_d^3)$, where q_d^i, is the proportion of approved transactions assigned to provider i on day d, with $d = 0, 1, 2, \ldots$. The system states are defined based on these performance measures. The system states are defined as:

State 1 State of the system is 1 if $q_1^d \geq q_1^{d-1}, q_2^d \geq q_2^{d-1}, q_3^d \geq q_3^{d-1}$,
State 2 if $q_1^d \geq q_1^{d-1}, q_2^d \geq q_2^{d-1}, q_3^d < q_3^{d-1}$,
State 3 if $q_1^d \geq q_1^{d-1}, q_2^d < q_2^{d-1}, q_3^d \geq q_3^{d-1}$,
State 4 if $q_1^d \geq q_1^{d-1}, q_2^d < q_2^{d-1}, q_3^d < q_3^{d-1}$,
State 5 if $q_1^d < q_1^{d-1}, q_2^d \geq q_2^{d-1}, q_3^d \geq q_3^{d-1}$,
State 6 if $q_1^d < q_1^{d-1}, q_2^d \geq q_2^{d-1}, q_3^d < q_3^{d-1}$,
State 7 if $q_1^d < q_1^{d-1}, q_2^d < q_2^{d-1}, q_3^d \geq q_3^{d-1}$, and
State 8 if $q_1^d < q_1^{d-1}, q_2^d < q_2^{d-1}, q_3^d < q_3^{d-1}$.

The states are identified with triplets of signatures as follows: state 1 with $(+++)$, state 2 with $(++-)$, state 3 with $(+-+)$, state 4 with $(+--)$, state 5 with $(-++)$, state 6 with $(+++)$, state 7 with $(--+)$, and state 8 with $(---)$. The action space consists of making a small perturbation to an allocation distribution at hand, restricted to only seven actions. If $p = (p_1, p_2, p_3)$ is an allocation probability vector, then seven probability vectors are derived from p, one according to each action. The actions are described as follows, using the perturbation parameter δ, a prefixed small positive number.

Action 1: leaves p unchanged.

Action 2: The perturbed $\bar{p}$ is given by $\bar{p}_1 = p_1 + \delta\, p_3 p_1/(p_1 + p_2), \overline{p_3} = (1-\delta)\, p_3$, and $\bar{p}_2 = 1 - \bar{p}_1 - \bar{p}_3$.

Action 3: $\bar{p}_1 = p_1 + \delta\, p_2 p_1/(p_1 + p_3), \overline{p_2} = (1-\delta)\, p_2$, and $\bar{p}_3 = 1 - \bar{p}_1 - \bar{p}_2$

Action 4: $\overline{p_1} = (1-\delta)\, p_1$, $\overline{p_2} = p_2 + \delta\, p_2 p_1/(p_2 + p_3)$ and $\bar{p}_3 = 1 - \bar{p}_1 - \bar{p}_2$.

Action 5: $\overline{p_1} = (1 - \frac{\delta}{2})\, p_1, \overline{p_3} = (1 - \frac{\delta}{2})\, p_3$ and $\bar{p}_2 = 1 - \bar{p}_1 - \bar{p}_3$.

Action 6: $\overline{p_1} = \left(1 - \frac{\delta}{2}\right) p_1, \overline{p_2} = \left(1 - \frac{\delta}{2}\right) p_2$, and $\bar{p}_3 = 1 - \bar{p}_1 - \bar{p}_2$.

Action 7: $\overline{p_2} = \left(1 - \frac{\delta}{2}\right) p_2$, $\overline{p_3} = \left(1 - \frac{\delta}{2}\right) p_3$, and $\bar{p}_1 = 1 - \bar{p}_2 - \bar{p}_3$.

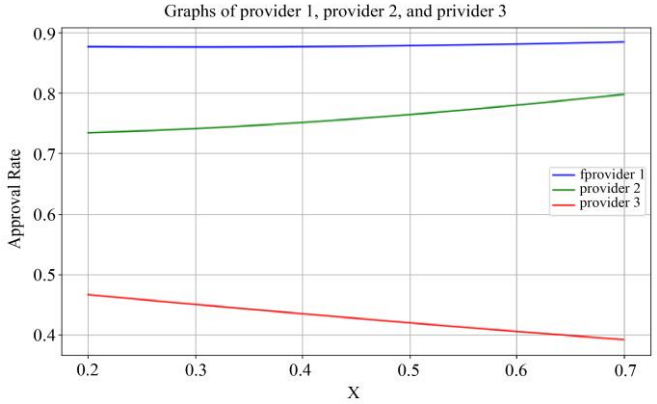


Fig. 2 Provider Performance Curves. The parameters of the logistic function are: : $a_1 = 2$, $b_1 = -1$, $c_1 = 0.3$, $a_2 = 1$, $b_2 = -0.9$, $c_2 = 0.6$ and $a_3 = 0$, $b_3 = -0.3$, $c_3 = 0.9$

To analyze the system and the strategies, the approval proportions are required and are simulated. The approval rate q_i of provider, i is assumed to be a function of p_i, the proportion of transactions assigned. Since q_i values are between 0 and 1, logistic functions of the form: $q_i(p) = exp(a_i + b_i p + c_i p^2) / (1 + exp(a_i + b_i p + c_i p^2))$ are used. Here $a_i, b_i, and\ c_i$ are chosen differently for each provider i, where $i = 1,2,3$. The choice of these parameters and the corresponding graphs of the functions are shown in Figure. The number of transactions each day, N_d, also needs to be simulated. For simplicity, N_d values are randomly simulated from the interval 1000 to 3000. The formula for the reward for day d action (sum of the rewards from the three providers) is given by: $N_d \sum_{i=1}^{3} p_{d+1}^i q_d^i$.

5.4. Understanding the Extent of Action Space

As the number of days in the planning horizon increases, the number of actions grows exponentially. This can be illustrated as follows: at the beginning of day 0, p^0 is the previous allocation, and q_{-1} represents the corresponding approval proportions. Applying the 7 actions to p^0 results in 7 new allocation distributions and their corresponding approval proportions. Comparing these approval proportions with q_{-1} will determine the initial states of day 1, potentially leading to 7 distinct allocation distributions at the beginning of day 1. Applying the 7 actions to each of these distributions results in up to 49 possible initial distributions at the beginning of day 2. Continuing this calculation, the number of possible initial distributions at the beginning of day d is 7^d. Handling this exponential growth is challenging. This is where the dynamic programming approach proves beneficial. The algorithm for obtaining the optimal solution under the considered action space will be illustrated with an example. For the purpose of illustration, the planning horizon is restricted to $T = 1$, meaning the system concludes at the end of day 1 or, equivalently, at the beginning of day 2. The objective is to determine the actions that lead to the maximum total reward.

5.4.1. Example

Consider the case where $T = 2, N_0 = 2400, and\ p_0 = (0.370, 0.365, 0.265)$. From p_0, the approval rates $q_{-1} = (0.84, 0.68, 0.5)$ are computed. The result of applying the 7 actions to p_0 are summarized in Table 2. Starting with $p_0 = (0.370, 0.365, 0.265)$ and applying action 2, the providers are assigned transactions in the proportions $p_1 = (0.383, 0.378, 0.239)$ (as shown in the row corresponding to action 2 in the table). The approval proportions for p_1 are $q_0 = (0.84, 0.678, 0.495)$. Comparing these with q_{-1}, the resulting state of the system at the beginning of day 1 is 8. The corresponding reward for day 0 from action 2 is 1671. The results of other actions can be observed in the table. Notably, actions 1 and 6 lead to the same state, namely, state 1. However, comparing the corresponding rewards, 1658 and 1639, action 1 is preferable to action 6 for reaching state 1 on day 1.

Table 2. Results of applying the 7 actions on day 1

	Resulting allocation			Resulting approval rate				
A	$\bar{p_1}$	$\overline{p_2}$	$\overline{p_3}$	$\bar{q_1}$	$\overline{q_2}$	$\overline{q_3}$	RS	TR
1	0.37	0.37	0.27	0.84	0.68	0.50	1	1658
2	0.38	0.38	0.24	0.84	0.68	0.50	8	1671
3	0.39	0.33	0.28	0.84	0.68	0.50	5	1662
4	0.33	0.39	0.28	0.85	0.68	0.50	3	1639
5	0.35	0.40	0.25	0.84	0.68	0.50	4	1655
6	0.35	0.35	0.30	0.84	0.68	0.50	1	1639
7	0.40	0.35	0.25	0.84	0.68	0.50	6	1673

Note : A: action, RS: resulting state, TR: total rewards

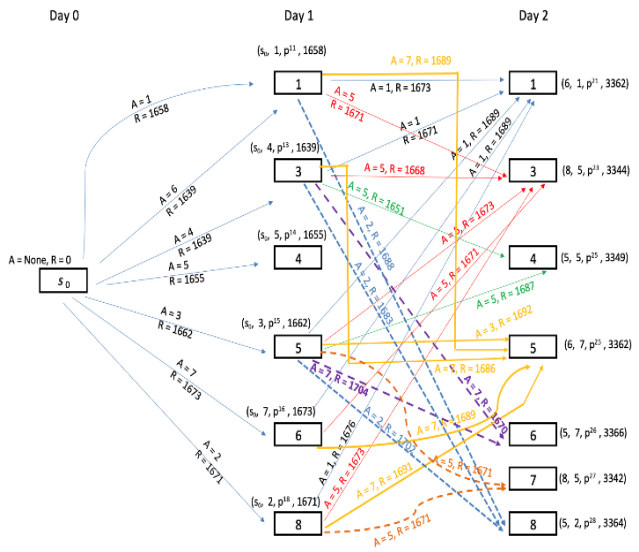


Fig. 3 Schematic procedure for optimal solution. The p^{ds} is the allocation distribution of state s of Day d. The p^{27} is obtained by applying action 5 to p^{18}. The edges from state 4 of Day 1 to other states of Day 2 are not shown for better presentation. Dropping these edges does not impact the computation of the rewards of states of Day 2 as they are dominated by others while computing reward attributes of states of Day 2.

5.5. Algorithm for Obtaining Optimal Solution

The algorithm is based on Bellman's optimality principle, which states that an optimal policy has the property that, regardless of the initial state and initial decision, the subsequent decisions must constitute an optimal policy with respect to the state resulting from the first decision [12]. The algorithm will be described using the example provided earlier. Refer to Figure 13 for cross-referencing. A graph with nodes and edges represents the system [13] [14]. Nodes represent the states of the system on different days, while the edges represent the actions. The rewards of the actions, which depend on the initial state and the action taken, are indicated on the edges along with the actions.

5.5.1. Algorithm

Step 1 Start with s_0 of Day 1. Set its action as "None" and its reward as $R = 0$.

Step 2 Apply action a on p^0 and determine the modified allocation p and set it as $p^1(a)$, call it $q_0(a)$, and determine the state (s_a) comparing it with q_{-1}. Compute the reward $R(s_0, a)$. Create an edge-connecting s_0 *and* s_a and label the edge with action a and reward $R(s_0, a)$.

Step 3 Repeat Step 2 for all 7 actions and prepare a list of states for Day 1. The list of states for the example is shown under Day 1 in the Figure 13. Note there are only six states in Day 1 as actions 1 and 6 lead to the same state 1.

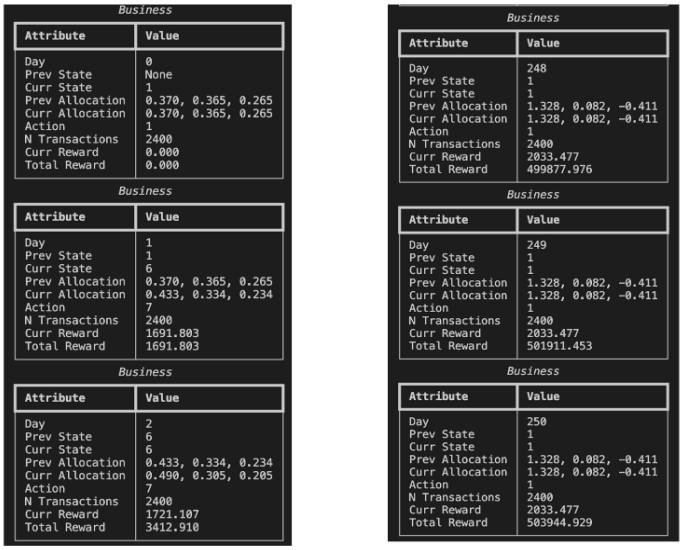
Step 4 Label the states of Day 1 with four attributes (s_0, a, p^{1s}, r) as follows. Consider a state s_1 of Day 1. List all actions on p^0 that led to state s_1. Suppose these actions are $a_1, a_2, \ldots, a_k$. Suppose j is such that $R(s_0, a_j) = max_{1 \le i \le k} R(s_0, a_i)$. If there are ties, break it arbitrarily. The four attributes of s_1 are: the first one is s_0, the second one is a_j, the third is p^{1s_1} and the fourth is $R(s_0, a_j)$. For the example from Figure 13, consider $s_1 = 1$. For this $k = 2, a_1 = 1, a_2 = 6$ *and* $j = 1$ *with* $R(s_0, a_1). = 1658$. Thus, state 1 is labelled with $(s_0,\ 1,\ p^{1s_0}, 1685)$. Note that p^{1s_0} is the allocation distribution obtained by applying action 1 on s_0. All other states of day 1 (3, 4, 5, 6 and 8) have only one action leading to them, and therefore, their reward attributes are the same as those of action rewards. The labels of these states are shown in Figure 13.

Step 5 The general computational steps for day d where $d \geq 2$. Suppose all states up to day d-1 are labelled. The procedure for labelling the states of day d with the four attributes mentioned previously is explained. Firstly, all 7 actions are applied to all states of day $d - 1$, generating a list of all possible states for day d. Consider any state s_d of day d. Let $a_1, a_2, \ldots, a_k$ be the actions leading to s_d. Let r_{si} be the reward attribute of the tail node of a_i, where $i = 1, 2, \ldots, k$. Let r_{ai} be the action reward of the action a_i on its tail node. Let j be such that:

$$r_{sj} + r_{aj} \geq r_{si} + r_{ai} \; for \; i = 1, 2, \ldots, k. \qquad (1)$$

Set the attributes of s_d as follows: the first attribute is the tail node of a_j, the second attribute is a_j, and the third attribute is p^{ds_d}, the allocation distribution resulting from a_j on its tail node. The fourth attribute of s_d *is* $r_{sj} + r_{aj}$. As an illustration, consider $d = 2$ (day 2) and $s_d = 8$. There are four actions, $a_1 = 2, a_2 = 2, a_3 = 2, a_4 = 2$ (in Figure 13, edges from state 4 of day 1 to states of day 2 are not shown; refer to the caption for further clarification) leading to state 8. Note that all these actions are equal to 2 (see the figure) but have different tail nodes. The tail nodes of a_1, a_2, a_3, a_4 are 1,3,4, and 5, respectively. Additionally, $r_{s1} = 1658, r_{a1} = 1688, r_{s2} = 1639, r_{a2} = 1683, r_{s3} = 1655, r_{a3} = 1688, r_{s4} = 1662, and\ r_{a4} = 1702.$ Here, $j = 4$, meaning that action 2 from node 5 of day 1 maximizes $r_{si} + r_{ai}$ with $r_{s4} + r_{a4} = 1662 + 1702 = 3364$. Therefore, the attributes of state 8 of day 2 are $(5, 2, p_{28}, 3364)$. This process of labelling states continues until all states of day T are completed.

Step 6 Identify the state of day T with the maximum reward attribute. This value is the optimum objective for the problem. The sequence of optimal actions, representing the optimal solution, is traced back from the optimal state of day T. For the example with $T = 2$, the optimal state is 8 with an optimal total reward of 3364. The optimal solution is traced back as follows: state 8 of day 2 is reached from state 5 with action 2. This information is found in the label of state 8 of day 2. From the label of state, 5 of day 1, state 5 is reached from s_0 with action 3.



Business

Attribute	Value
Day	0
Prev State	None
Curr State	1
Prev Allocation	0.370, 0.365, 0.265
Curr Allocation	0.370, 0.365, 0.265
Action	1
N Transactions	2400
Curr Reward	0.000
Total Reward	0.000

Business

Attribute	Value
Day	1
Prev State	1
Curr State	6
Prev Allocation	0.370, 0.365, 0.265
Curr Allocation	0.433, 0.334, 0.234
Action	7
N Transactions	2400
Curr Reward	1691.803
Total Reward	1691.803

Business

Attribute	Value
Day	2
Prev State	6
Curr State	6
Prev Allocation	0.433, 0.334, 0.234
Curr Allocation	0.490, 0.305, 0.205
Action	7
N Transactions	2400
Curr Reward	1721.107
Total Reward	3412.910

Business

Attribute	Value
Day	248
Prev State	1
Curr State	1
Prev Allocation	1.328, 0.082, −0.411
Curr Allocation	1.328, 0.082, −0.411
Action	1
N Transactions	2400
Curr Reward	2033.477
Total Reward	499877.976

Business

Attribute	Value
Day	249
Prev State	1
Curr State	1
Prev Allocation	1.328, 0.082, −0.411
Curr Allocation	1.328, 0.082, −0.411
Action	1
N Transactions	2400
Curr Reward	2033.477
Total Reward	501911.453

Business

Attribute	Value
Day	250
Prev State	1
Curr State	1
Prev Allocation	1.328, 0.082, −0.411
Curr Allocation	1.328, 0.082, −0.411
Action	1
N Transactions	2400
Curr Reward	2033.477
Total Reward	503944.929

Fig. 4 Python Output. The left panel presents details up to the first three nodes of the optimal path, while the right panel shows the last three nodes.

The optimal allocations are $p_{15} = (0.391, 0.329, 0.280)$ on day 0 (obtained by applying action 3 to $p_0 = (0.37, 0.365, 0.265)$) and (0.407, 0.341, 0.252) (obtained by applying action 2 to p_{15}). Thus, the optimal sequence of actions, given the initial state p_0, is (3,2), resulting in a maximum total reward of 3364.

5.5.2. Python Program

A Python program was developed to implement the algorithm for payment optimization using a dynamic programming approach. This program facilitates the evaluation of different strategies. Initially, the objective was to compare two strategies: revising transaction allocation daily versus revising it once every two weeks. The program is designed to allow for more flexible options, such as revising the allocation every two days, every three days, and so on. To illustrate the program's output, a sample output is presented in Figure 14.

5.6. Results

Using the provided Python code, a comparison of strategies is conducted. It is important to note that the optimal reward is achieved with the daily strategy, which involves changing the traffic allocation to providers every day. This reward is always greater than or equal to the optimal reward achieved with strategies where the traffic allocation is changed at longer intervals. This is because the set of options available under the less frequent strategies is a subset of those available under the daily strategy. The model and formulation considered in this paper quantify the magnitude of the gap between the optimal rewards. To compare different strategies, the number of transactions is kept constant at 2400. Within a total planning horizon of 250 days, results were compared across five different strategies: revising every day, revising once every five days, revising once every ten days, revising once every fifteen days, and revising once every twenty-five days. The results, presented in Table 3, indicate a significant improvement of 8.1%.

6. Conclusion

The payments optimization problem can significantly enhance transaction approvals for businesses, thereby increasing cash flow and improving the chances of transaction approval on the first attempt. This, in turn, reduces additional provider costs associated with retrying transactions in case of failures. Smart payments, where the system learns to take actions based on the current state, represent an advanced capability that many businesses aim to achieve. Solutions for smart payments typically involve reinforcement learning models, such as multi-armed bandits or contextual bandits, which require substantial setup engineering costs and have a steep learning curve.

Alternatives to these models include supervised learning solutions, which often demand extensive upkeep due to data drift and continuous modelling. This paper presents a novel dynamic programming approach to modelling the payments optimization problem, demonstrated through a simulated use case. The provided code is designed to be simple yet effective, allowing experimentation to determine the best initial state and optimal actions for each state to maximize transaction approval rates. The effectiveness of this approach is demonstrated by comparing different strategies; the results show that revising the traffic allocation daily can improve the overall reward by 8.1% for the considered simulated data.This approach offers a more accessible and maintainable alternative to traditional reinforcement learning models for businesses seeking to optimize their payment systems.

References

[1] Howard Anton, and Chris Rorres, *Elementary Linear Algebra*, 9th Ed., John Wiley & Sons, 1987. [Google Scholar] [Publisher Link]

[2] Amitav Banerjee, and Suprakash Chaudhury, "Statistics without Tears: Populations and Samples," *Industrial Psychiatry Journal*, vol. 19, no. 1, pp. 60-65, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[3] Ilker Etikan, and Kabiru Bala, "Sampling and Sampling Methods," *Biometrics & Biostatistics International Journal*, vol. 5, no. 6, pp. 215-217, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[4] Julius Hokka, "*Controlled Experiments for Data-Driven Retail Optimization*," Master's Thesis, 2023. [Google Scholar] [Publisher Link]

[5] Kirthi Kalyanam et al., "Cross Channel Effects of Search Engine Advertising on Brick & Mortar Retail Sales: Meta Analysis of Large Scale Field Experiments on Google.Com," *Quantitative Marketing and Economics*, vol. 16, pp. 1-42, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[6] Ron Kohavi et al., "Online Controlled Experiments at Large Scale," *KDD '13: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Chicago Illinois USA, pp. 1168-1176, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[7] Rembrand Koning, Sharique Hasan, and Aaron Chatterji, "Experimentation and Start-Up Performance: Evidence from A/B Testing," *Management Science*, vol. 68, no. 9, pp. 6434-6453, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[8] Sharon L. Lohr, *Sampling: Design and Analysis*, Chapman and Hall/CRC, 3rd ed., pp. 1- 674, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[9] Douglas C. Montgomery, *Design and Analysis of Experiments*, John Wiley & Sons, pp. 1-734, 2017. [Google Scholar] [Publisher Link]

[10] Douglas C. Montgomery, Cheryl L. Jennings, and Murat Kulahci, *Introduction to Time Series Analysis and Forecasting*, John Wiley & Sons, pp. 1-672, 2015. [Google Scholar] [Publisher Link]

[11] Katta G. Murty, *Linear Programming*, Springer, pp. 1-231, 1983. [Google Scholar] [Publisher Link]

[12] Thomas H Cormen et al., *Introduction to Algorithms*, 2nd Ed., MIT Press & McGraw-Hill, pp. 1-1180, 2001. [Google Scholar] [Publisher Link]

[13] Norman Biggs, E. Keith Lloyd, and Robin J. Wilson, *Graph Theory, 1736 – 1936*, Springer, pp. 1-239, 1986. [Google Scholar] [Publisher Link]

[14] Edward A. Bender, and S. Gill Williamson, *Decisions and Graphs*, S. Gill Williamson, 2010. [Google Scholar] [Publisher Link]

[15] Erhan Çınlar, and Norman J. Sollenberger, *Introduction to Stochastic Processes*, Waveland Press, pp. 1-402, 1986. [Google Scholar] [Publisher Link]

[16] Sheldon M. Ross, *Introduction to Probability Models*, Academic Press, pp. 1-784, 2014. [Google Scholar] [Publisher Link]

Appendix 1

Provider Performance Curves

```
import numpy as np
import plotly.graph_objs as go

# Define the logistic functions
def logistic_function(p, f, c):
    return c + np.exp(f) / (1 + np.exp(f))


def f1_f2_f3(p: np.ndarray) -> np.ndarray:
    """
    Compute f1, f2, f3 based on the input array p of size 3.

    Args:
    p (np.ndarray): Input array of size 3.

    Returns:
    np.ndarray: Output array of size 3 containing f1, f2, f3.
```

```
    """
    f1 = -1 + 5 * p[0] + 50 * (p[0] - 0.4)**2
    f2 = -1 + 5 * p[1] - 50 * np.abs(p[1] - 0.4)**1.2
    f3 = 0.7 - 50 * (p[2] - 0.4)**2

    return np.array([f1, f2, f3])

# Define the constants
# c = np.array([0.1, 0.3, 0.08])
c = np.array([0,0,0])

# Generate a range of p values
p_values = np.linspace(0, 1, 100)
p_vectors = np.vstack((p_values, p_values, p_values)).T

# Compute the logistic function values for each p
q_values = np.apply_along_axis(lambda p: logistic_function(p, f1_f2_f3(p), c), 1, p_vectors)

# Create traces for each logistic function
trace1 = go.Scatter(x=p_values, y=q_values[:, 0], mode='lines', name='Provider 1',
line=dict(color='blue'))
trace2 = go.Scatter(x=p_values, y=q_values[:, 1], mode='lines', name='Provider 2',
line=dict(color='red'))
trace3 = go.Scatter(x=p_values, y=q_values[:, 2], mode='lines', name='Provider 3',
line=dict(color='green'))

# Create the layout
layout = go.Layout(
    title=dict(
        text= “Provider Performance Curves”,
        x=0.5,
        xanchor='center’
    ),
    # xaxis range = range=[0.175, 0.75]
    xaxis=dict(title="Proportion of Transactions (p)", range = [0.2,0.6]),
    # yaxis range=[0.27, 1.1]
    yaxis=dict(title="Approval Rate (q)"),
    showlegend=True
)

# Create the figure
fig = go.Figure(data=[trace1, trace2, trace3], layout=layout)

# Show the plot
fig.show()
```

Test $i.i.d$ assumption

```
import numpy as np

# Set the random seed for reproducibility
RANDOM_SEED = 2023
np.random.seed(RANDOM_SEED)

# Function to simulate a sequence satisfying hypothesis H (iid sequence)
def simulate_iid_sequence(length, distribution):
    """
    Simulate a sequence of given length that satisfies hypothesis H.

    Args:
    length (int): Length of the sequence.
    distribution (list): Pre-specified distribution.

    Returns:
    np.array: Simulated iid sequence.
    """
    return np.random.choice([1, 2, 3], size=length, p=distribution)

# Function to simulate a sequence violating hypothesis H (Markov chain sequence)
def simulate_markov_sequence(length, transition_matrix):
    """
    Simulate a sequence of a given length that violates hypothesis H using a Markov chain.

    Args:
    length (int): Length of the sequence.
    transition_matrix (np.array): One-step transition matrix.

    Returns:
    np.array: Simulated Markov sequence.
    """
    sequence = np.zeros(length, dtype=int)
    # Initialize the first state randomly
    sequence[0] = np.random.choice([1, 2, 3])

    for i in range(1, length):
        current_state = sequence[i-1]
        # Choose the next state based on the transition matrix
        sequence[i] = np.random.choice([1, 2, 3], p=transition_matrix[current_state-1])

    return sequence
```

```
# Length of the sequences
length = 2000

# Pre-specified distribution p
p = [0.7, 0.2, 0.1]

# Transition matrix to simulate a sequence violating hypothesis H
transition_matrix = np.array([
    [0.75, 0.10, 0.15],
    [0.70, 0.20, 0.10],
    [0.70, 0.15, 0.15]
])

# Simulate sequences
sequence_iid = simulate_iid_sequence(length, p)
sequence_markov = simulate_markov_sequence(length, transition_matrix)

# Display the first 10 elements of each sequence as a sample
print("First 10 elements of the IID sequence:", sequence_iid[:10])
print("First 10 elements of the Markov sequence:", sequence_markov[:10])
```

 # Plot Autocorrelation functions

```
import numpy as np
import plotly.graph_objs as go
from scipy.signal import correlate

# Function to compute and plot autocorrelation
def plot_autocorrelation(sequence, title):
    # Compute autocorrelation
    autocorr = correlate(sequence - np.mean(sequence), sequence - np.mean(sequence),
mode='full')
    autocorr = autocorr[autocorr.size // 2:]
    autocorr /= autocorr[0]

    # Compute the standard deviation for the sequence
    std_dev = np.std(sequence)
    conf_interval = 1.96 / np.sqrt(len(sequence))

    # Create the plot
    trace = go.Scatter(
        x=np.arange(len(autocorr)),
        y=autocorr,
        mode='markers+lines',
```

```
        name='ACF'
    )

    # Create the lines for ±1.96 standard deviations
    upper_bound = go.Scatter(
        x=np.arange(len(autocorr)),
        y=np.ones(len(autocorr)) * conf_interval,
        mode='lines',
        line=dict(color='red', dash='dash'),
        name='+1.96 SD'
    )

    lower_bound = go.Scatter(
        x=np.arange(len(autocorr)),
        y=-np.ones(len(autocorr)) * conf_interval,
        mode='lines',
        line=dict(color='red', dash='dash'),
        name='-1.96 SD'
    )

    layout = go.Layout(
        title=dict(
            text=title,
            x=0.5,  # Center the title
            xanchor='center'
        ),
        xaxis=dict(title='Lag'),
        yaxis=dict(title='Autocorrelation'),
        showlegend=False
    )

    fig = go.Figure(data=[trace, upper_bound, lower_bound], layout=layout)
    fig.show()

# Plot autocorrelation for both sequences
plot_autocorrelation(sequence_iid[:30], 'Autocorrelation of i.i.d Sequence')
plot_autocorrelation(sequence_markov[:30], 'Autocorrelation of non i.i.d Sequence')
```

Chi – Square Statistic

```
import numpy as np
from collections import Counter
```

```
# Pre-specified distribution p
p = [0.7, 0.2, 0.1]

def compute_transition_matrix(sequence):
    """
    Compute the transition matrix from the given sequence.

    Args:
    sequence (list or np.array): Sequence of states.

    Returns:
    np.array: Transition matrix.
    """
    transitions = Counter((sequence[i], sequence[i+1]) for i in range(len(sequence) - 1))
    total_transitions = {k: sum(v for kk, v in transitions.items() if kk[0] == k) for k in
range(1, 4)}

    transition_matrix = np.zeros((3, 3), dtype=int)

    for (i, j), count in transitions.items():
        transition_matrix[i-1, j-1] = count

    return transition_matrix, total_transitions

def compute_expected_transitions(total_transitions, p):
    """
    Compute the expected transitions based on the pre-specified distribution.

    Args:
    total_transitions (dict): Total transitions from each state.
    p (list): Pre-specified distribution.

    Returns:
    np.array: Expected transition matrix.
    """
    expected_transitions = np.zeros((3, 3))

    for i in range(3):
        for j in range(3):
            expected_transitions[i, j] = total_transitions[i+1] * p[j]

    return expected_transitions

def compute_chi_square(observed, expected):
    """
```

```
    Compute the chi-square statistic.

    Args:
    observed (np.array): Observed transition matrix.
    expected (np.array): Expected transition matrix.

    Returns:
    float: Chi-square statistic.
    """
    chi_square = np.sum((observed - expected)**2 / expected)
    return chi_square

# Compute transition matrices and totals for both sequences
observed_iid, total_transitions_iid = compute_transition_matrix(sequence_iid)
observed_markov, total_transitions_markov = compute_transition_matrix(sequence_markov)

# Compute expected transition matrices
expected_iid = compute_expected_transitions(total_transitions_iid, p)
expected_markov = compute_expected_transitions(total_transitions_markov, p)

# Compute chi-square statistics
chi_square_iid = compute_chi_square(observed_iid, expected_iid)
chi_square_markov = compute_chi_square(observed_markov, expected_markov)

# Print results
def print_matrix(title, matrix):
    print(f"{title}:")
    for row in matrix:
        print(" ".join(f"{val:8.1f}" for val in row))
    print()

# Print results
print_matrix("Observed Transitions (i.i.d)", observed_iid)
print_matrix("Expected Transitions (i.i.d)", expected_iid)
print(f"Chi-square Statistic (i.i.d): {chi_square_iid:.2f}\n")

print_matrix("Observed Transitions (non i.i.d)", observed_markov)
print_matrix("Expected Transitions (non i.i.d)", expected_markov)
print(f"Chi-square Statistic (non i.i.d): {chi_square_markov:.2f}")
```

Dynamic Programming

```
import dataclasses
import itertools
import math
```

```
import typing as t

import numpy as np
import rich
from rich.table import Table

DELTA = 0.1  # The amount by which the allocation changes in each action

# Type alias for a 3-element numpy array of floats
Allocation: t.TypeAlias = np.ndarray[t.Literal[3], np.dtype[np.float64]]
ApprovalRates: t.TypeAlias = np.ndarray[t.Literal[3], np.dtype[np.float64]]
SingleState: t.TypeAlias = t.Literal[1, -1]

@dataclasses.dataclass
class State:
    short: int
    expanded: tuple[SingleState, SingleState, SingleState]
    approval_rates: ApprovalRates

    STATE_MAP: t.ClassVar[dict[tuple[SingleState, SingleState, SingleState], int]] = {
        (1, 1, 1): 1,
        (1, 1, -1): 2,
        (1, -1, 1): 3,
        (1, -1, -1): 4,
        (-1, 1, 1): 5,
        (-1, 1, -1): 6,
        (-1, -1, 1): 7,
        (-1, -1, -1): 8,
    }
    REVERSE_STATE_MAP: t.ClassVar[
        dict[int, tuple[SingleState, SingleState, SingleState]]
    ] = {v: k for k, v in STATE_MAP.items()}

    @classmethod
    def from_short(cls, short: int, approval_rates: ApprovalRates) -> "State":
        return cls(short, cls.REVERSE_STATE_MAP[short], approval_rates=approval_rates)

    @classmethod
    def from_expanded(
        cls,
        expanded: tuple[SingleState, SingleState, SingleState],
        approval_rates: ApprovalRates,
    ) -> “State”:
        return cls(cls.STATE_MAP[expanded], expanded, approval_rates=approval_rates)

    @classmethod
    def from_approval_rates(
        cls, prev_approval_rates: ApprovalRates, curr_approval_rates: ApprovalRates
    ) -> “State”:
        assert len(prev_approval_rates) == len(curr_approval_rates) == 3
        return cls.from_expanded(
            tuple(  # type: ignore
                1 if qi >= pi else -1
                for pi, qi in zip(prev_approval_rates, curr_approval_rates)
            ),
            approval_rates=curr_approval_rates,
```

```
        )

@dataclasses.dataclass
class Business:
    day: int
    prev_state: State | None
    curr_state: State
    prev_allocation: Allocation | None
    curr_allocation: Allocation
    action: int
    n_transactions: int
    curr_reward: float
    prev_business: "Business | None" = None

    def __rich_repr__(self):
        table = Table(title="Business")
        table.add_column("Attribute")
        table.add_column("Value")
        table.add_row("Day", f"{self.day}")
        table.add_row(
            "Prev State", f"{self.prev_state.short if self.prev_state else None}"
        )
        table.add_row("Curr State", f"{self.curr_state.short}")
        if self.prev_allocation is not None:
            table.add_row(
                “Prev Allocation”,
                ", ".join([f"{x:.3f}" for x in self.prev_allocation]),
            )
        table.add_row(
            "Curr Allocation", ", ".join([f"{x:.3f}" for x in self.curr_allocation])
        )
        table.add_row("Action", f"{self.action}")
        table.add_row("N Transactions", f"{self.n_transactions}")
        table.add_row("Curr Reward", f"{self.curr_reward:.3f}")
        table.add_row("Total Reward", f"{self.total_reward:.3f}")
        return table

    @property
    def total_reward(self):
        return self.curr_reward + (
            self.prev_business.total_reward if self.prev_business else 0
        )

def get_approval_rates(allocation: Allocation) -> ApprovalRates:
    “““Calculate the approval rates for a given allocation of providers."""

    # Define the coefficients for the approval rate function per provider
    provider0 = (2.0, -1.0, 0.3)
    provider1 = (1.0, -0.9, 0.6)
    provider2 = (0.0, -0.3, 0.9)

    def approval_rate_for_provider(
        allocation: float, alpha: float, beta: float, gamma: float
    ) -> float:
        exp_term = np.exp(alpha + beta * allocation + gamma * allocation**2)
        return np.round(exp_term / (1 + exp_term), 3)
```

```
    return np.array(
        [
            approval_rate_for_provider(allocation[0], *provider0),
            approval_rate_for_provider(allocation[1], *provider1),
            approval_rate_for_provider(allocation[2], *provider2),
        ]
    )

def compute_new_state_allocation(action: int, allocation: Allocation, delta: float):
    # sourcery skip: extract-duplicate-method, move-assign-in-block, switch
    allocation = allocation.copy()
    if action == 1:
        return allocation
    elif action == 2:  # a=2 means (+,+,-)
        amount_to_adjust = allocation[2] * delta
        ratio = allocation[1] / (allocation[0] + allocation[1])
        allocation[2] = 0.9 * allocation[2]
        allocation[1] = allocation[1] + amount_to_adjust * ratio
        allocation[0] = 1 - allocation[1] - allocation[2]
        return allocation
    elif action == 3:  # a=3 means (+,-,+)
        amount_to_adjust = allocation[1] * delta
        ratio = allocation[2] / (allocation[0] + allocation[2])
        allocation[1] = 0.9 * allocation[1]
        allocation[2] = allocation[2] + amount_to_adjust * ratio
        allocation[0] = 1 - allocation[1] - allocation[2]
        return allocation
    elif action == 4:  # a=4 means (-,+,+)
        amount_to_adjust = allocation[0] * delta
        ratio = allocation[2] / (allocation[1] + allocation[2])
        allocation[0] = allocation[0] - amount_to_adjust
        allocation[2] = allocation[2] + amount_to_adjust * ratio
        allocation[1] = 1 - allocation[0] - allocation[2]
        return allocation
    elif action == 5:  # a=5 means (-,+,-)
        amount_to_adjust = (allocation[0] + allocation[2]) * delta
        ratio = 0.5
        allocation[1] = allocation[1] + amount_to_adjust
        allocation[2] = allocation[2] - amount_to_adjust * ratio
        allocation[0] = 1 - allocation[1] - allocation[2]
        return allocation
    elif action == 6:  # a=6 means (-,-,+)
        amount_to_adjust = (allocation[0] + allocation[1]) * delta
        ratio = 0.5
        allocation[2] = allocation[2] + amount_to_adjust
        allocation[0] = allocation[0] - amount_to_adjust * ratio
        allocation[1] = 1 - allocation[0] - allocation[2]
        return allocation
    elif action == 7:  # a=7 means (+,-,-)
        amount_to_adjust = (allocation[1] + allocation[2]) * delta / 2
        allocation[1] = allocation[1] - amount_to_adjust
        allocation[2] = allocation[2] - amount_to_adjust
        allocation[0] = 1 - allocation[1] - allocation[2]
        return allocation
    else:
```

```
        raise ValueError(f"Invalid action: {action}")

def transition(
    business: Business,
    action: int,
    delta: float,
    n_transactions: int | None = None,
    horizon: int = 1,
) -> Business:
    n_transactions = (
        business.n_transactions if n_transactions is None else n_transactions
    )
    # First day take action and compute the reward
    final_allocation = new_allocation = compute_new_state_allocation(
        action, business.curr_allocation, delta
    )
    new_approval_rates = get_approval_rates(new_allocation)
    final_state = new_state = State.from_approval_rates(
        business.curr_state.approval_rates, new_approval_rates
    )
    curr_reward = compute_total_reward(
        n_transactions, new_allocation, new_approval_rates
    )
    for _ in range(1, horizon):
        # For all remaining days, don't take any action
        new_allocation = compute_new_state_allocation(1, new_allocation, delta)
        new_approval_rates = get_approval_rates(new_allocation)
        new_state = state.from_approval_rates(
            new_state.approval_rates, new_approval_rates
        )
        curr_reward = curr_reward + compute_total_reward(
            n_transactions, new_allocation, new_approval_rates
        )
    return Business(
        day=business.day + horizon,
        prev_state=business.curr_state,
        curr_state=final_state,
        prev_allocation=business.curr_allocation,
        curr_allocation=final_allocation,
        action=action,
        n_transactions=n_transactions,
        curr_reward=curr_reward,
        prev_business=business,
    )

def compute_total_reward(
    n_transactions: int, allocation: Allocation, approval_rates: ApprovalRates
):
    return n_transactions * sum(allocation * approval_rates)

def sdp(
    businesses: list[Business],
    delta: float,
    n_transactions: int | None = None,
    debug: bool = False,
    horizon: int = 1,
```

```
) -> list[Business]:
    # sourcery skip: use-itertools-product
    business_day = businesses[0].day
    assert all(business.day == business_day for business in businesses)
    all_states = [x.curr_state.short for x in businesses]
    assert len(all_states) <= len(State.STATE_MAP)
    next_businesses = []
    for business in businesses:
        for action in range(1, 8):
            next_business = transition(
                business, action, delta, n_transactions, horizon=horizon
            )
            next_businesses.append(next_business)
    grouped_by_state = itertools.groupby(
        sorted(next_businesses, key=lambda b: b.curr_state.short),
        key=lambda b: b.curr_state.short,
    )
    grouped_by_state = [(key, list(group)) for key, group in grouped_by_state]
    display_debug_info(debug, business_day, grouped_by_state)
    result = {
        key: max(group, key=lambda b: b.total_reward) for key, group in grouped_by_state
    }
    return list(result.values())

def display_debug_info(
    debug: bool, business_day: int, grouped_by_state: list[t.Tuple[int, list[Business]]]
):
    table = Table(title=f"Debug Day: {business_day}")
    table.add_column("Group State")
    table.add_column("Curr State")
    table.add_column("Action")
    table.add_column("Prev State")
    table.add_column("Curr Allocation")
    table.add_column("Curr Approval Rates")
    table.add_column("Curr Reward")
    table.add_column("Total Reward")
    for the state, group in grouped_by_state:
        max_reward = max(b.curr_reward for b in group)
        for business in the group:
            style = (
                "bold green" if math.isclose(business.curr_reward, max_reward) else ""
            )
            table.add_row(
                f"{state}",
                f"{business.curr_state.short}",
                f"{business.action}",
                f"{business.prev_state.short}",  # type: ignore
                ", ".join([f"{x:.3f}" for x in business.curr_allocation]),
                ", ".join([f"{x:.3f}" for x in business.curr_state.approval_rates]),
                f"{business.curr_reward:.3f}",
                f"{business.total_reward:.3f}",
                style=style,
            )
        table.add_row(
            "---",
            "---",
```

```
            "---",
            "---",
            "---",
            "---",
            "---",
        )
    if debug:
        rich.print(table)

def main(n_days: int, seed: int = 42, horizon: int = 1, debug: bool = False) -> float:
    rng = np.random.default_rng(seed)
    assert n_days % horizon == 0, "n_days must be divisible by horizon"
    businesses = [
        Business(
            day=0,
            prev_state=None,
            curr_state=State.from_approval_rates(
                np.r_[0, 0, 0], np.r_[0.842, 0.679, 0.496]
            ),
            prev_allocation=np.r_[0.37, 0.365, 0.265],
            curr_allocation=np.r_[0.37, 0.365, 0.265],
            action=1,
            n_transactions=2400,
            curr_reward=0,
        )
    ]
    n_transactions = 2400
    print("Day 0")
    print(f"Curr Allocation: {businesses[0].curr_allocation}")
    print(f"Curr Approval Rates: {businesses[0].curr_state.approval_rates}")
    print("N transactions", businesses[0].n_transactions)
    print("Current state", businesses[0].curr_state.short)
    for day in range(n_days // horizon):
        businesses = sdp(
            businesses,
            DELTA,
            n_transactions=n_transactions,
            debug=debug,
            horizon=horizon,
        )
        max_reward = max(business.total_reward for business in businesses)
        table = Table(title=f"Day: {(day + 1) * horizon}")
        table.add_column("Curr State")
        table.add_column("Action")
        table.add_column("Prev State")
        table.add_column("Curr Allocation")
        table.add_column("Curr Approval Rates")
        table.add_column("Curr Reward")
        table.add_column("Total Reward")
        for business in businesses:
            style = (
                "bold green" if math.isclose(business.total_reward, max_reward) else ""
            )
            table.add_row(
                f"{business.curr_state.short}",
                f"{business.action}",
```

```
                f"{business.prev_state.short}",  # type: ignore
                ", ".join([f"{x:.3f}" for x in business.curr_allocation]),
                ", ".join([f"{x:.3f}" for x in business.curr_state.approval_rates]),
                f"{business.curr_reward:.3f}",
                f"{business.total_reward:.3f}",
                style=style,
            )
        rich.print(table)
        n_transactions = 2400  # rng.integers(1000, 3000)
        # n_transactions = rng.integers(1000, 3000)
    best_business = max(businesses, key=lambda b: b.total_reward)
    print("Business Trace")
    best_reward = best_business.total_reward
    trace = [best_business]
    while best_business:
        best_business = best_business.prev_business
        if best_business:
            trace.append(best_business)
    for business in reversed(trace):
        rich.print(business.__rich_repr__())
    return best_reward

if __name__ == "__main__":
    horizon = int(input("Enter the horizon: "))
    main(250, horizon=horizon)
```