Finite State Machine and Matrix Representation for Solving Transition Modeling for Dynamic Programming Solution

Ryan Garnet Andrianto, Rully Soelaiman, *Member, IAENG,* and Misbakhul Munir Irfan Subakti

***Abstract*—Efficient computation of expected values is paramount in scenario analysis and decision-making, especially for problems involving large finite state machines with complex state dependencies and transitions. Traditional approaches relying on linear equation systems often fall short under such demanding conditions. Addressing this critical need, the proposed method integrates finite state machines to model transitions and matrices to manage state properties, delivering a breakthrough in efficiency. The matrix transition model, enhanced with an optimized memory management technique, achieves an average computation time of just 0.179 seconds—over 22 times faster than the strict 4-second time limit set by the problem setter—and consumes only 5.63 MB of memory, a mere 0.36% of the 1536 MB limit. These results underscore the solution's exceptional capability to not only meet but vastly exceed stringent performance requirements, redefining expectations for large-scale finite state machine calculations.**

***Index Terms*—expected value, geometric series, recurrence relations, memory optimization.**

I. Introduction

FINITE state machines are computational models extensively utilized for simulating sequential logic. These models serve as versatile tools for addressing problems in diverse domains, including mathematics, artificial intelligence, gaming, and linguistics [1]. In practical applications, finite state machines are commonly employed to represent and manage the states of dynamic systems. This paper leverages the concept of a cookie crane as a representative example to construct and analyze a finite state machine. The cookie crane is chosen for its central role in the Sphere Online Judge (SPOJ) problem TAP2015E, which serves as the benchmark for validating the proposed methodology.

To introduce the concept, a single cookie crane is initially considered. Fig. 1 illustrates three distinct states of a single cookie crane: (a) the crane is empty and carries no cookies; (b) it carries one cookie; and (c) it carries two cookies. The state of the crane evolves as it picks up and packages cookies. In this paper, the crane's state is denoted as $\{x\}$, where x represents the number of cookies currently carried. Array notation is adopted for state representation, facilitating

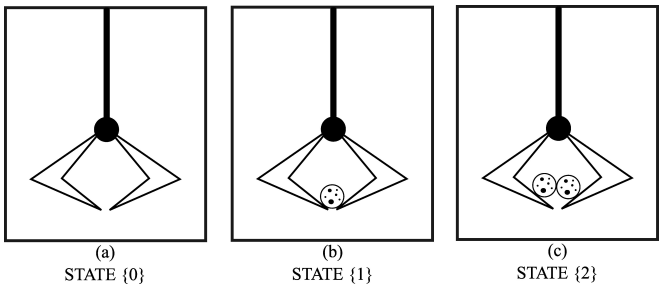

Fig. 1. Illustration of three distinct states of the cookie crane.

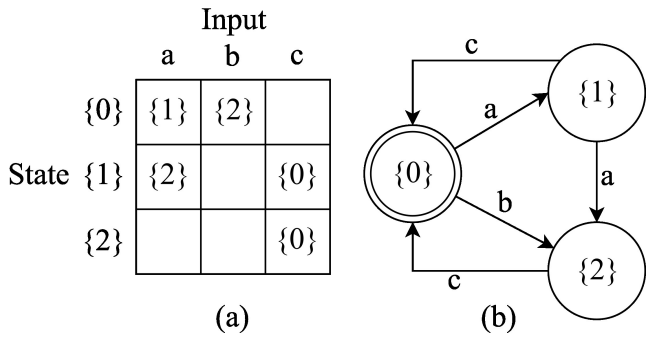

Fig. 2. Matrix representation and finite state machine of the cookie crane.

extension to systems involving multiple cranes operating in coordination.

State transitions occur as the cookie crane picks up cookies, with each transition's feasibility governed by problem-specific rules or constraints. A simple example is provided to elucidate this concept. Fig. 1 visually represents the cookie crane's states, illustrating its physical operation. However, computational analysis necessitates a mathematical representation of these transitions. In this work, the crane is modeled as a finite state machine and represented using a transition matrix, as depicted in Fig. 2. This mathematical abstraction enables efficient analysis and computation.

Fig. 2 contains two illustrations: (a) a matrix representing state transitions and (b) a finite state machine diagram for the cookie crane. In the finite state machine, state $\{0\}$ denotes the initial state. The selection of the starting state may vary based on contextual requirements; here, $\{0\}$ represents an empty crane, consistent with the initial state defined in the SPOJ TAP2015E problem.

State transitions are driven by actions such as picking up cookies, with each action occurring probabilistically to reflect real-world uncertainty. To effectively manage these probabil-

Manuscript received June 17, 2024; revised December 20, 2024. This work was supported in part by Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia.

Ryan Garnet Andrianto is a software engineer at PT Informasi Teknologi Indonesia, Jakarta Selatan, Indonesia, and is a graduate of Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia (e-mail: ryangarnetandrianto@gmail.com).

Rully Soelaiman is an associate professor at Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia (e-mail: rully130270@gmail.com).

Misbakhul Munir Irfan Subakti is an assistant professor at Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia (e-mail: yifana@gmail.com).

ities and transitions, a matrix representation is employed. The matrix facilitates efficient computation through operations such as addition and multiplication, which underpin the analysis detailed in subsequent sections.

In finite state machines, each state depends on its predecessor, except for the initial state. For instance, state $\{2\}$ is reachable from state $\{0\}$ via two actions. This dependency introduces recurrence relationships among states, necessitating precise calculations for expected values in state-dependent systems. Traditional approaches face challenges in handling such dependencies efficiently due to exponential growth in computational complexity with the number of actions N. This paper addresses these challenges by focusing on the expected number of cookies packaged by the crane after N actions. The proposed method overcomes complexity barriers, enabling computations that would otherwise be infeasible.

The expected value is a fundamental concept in statistics, representing the central tendency of a probability distribution [1]. It serves as a predictive measure for evaluating outcomes and supports decision-making processes in uncertain environments [3]. In this study, the objective is to calculate the expected number of cookies packaged by the crane after performing N actions. While dynamic programming offers a viable approach to handle state dependencies, the computational complexity increases significantly as N grows due to the expanding sequence of dependencies. This highlights the critical need for more efficient algorithms to address the rising resource demands effectively. The efficient algorithm proposed in this paper addresses this challenge by utilizing finite state machine and matrix representations to model transitions for dynamic programming solutions.

The SPOJ TAP2015E – Perfect Packing problem serves as the primary case study, presenting a critical challenge of determining the number of cookies a factory can produce while adhering to probabilistic actions and strict packaging rules [2]. Efficient solutions to this problem are essential due to the high computational complexity involved in handling state dependencies and transitions. The proposed approach, which integrates finite state machines and matrix representations, delivers a groundbreaking improvement in computational efficiency. This solution achieves an average computation time of just 0.179 seconds, dramatically outperforming the 4-second time limit and exceeding the problem setter's expectations by 22 times. The method demonstrates robust performance, effectively managing up to 210 fully connected states in the worst-case scenario. Furthermore, this solution ranks as the top submission for the given problem on the SPOJ platform, highlighting the critical need for such innovative methods and underscoring the transformative potential of the proposed approach.

The remainder of this paper is structured as follows: Section II introduces the finite state machine and matrix transition methodology. Section III presents the experimental results and analysis. Finally, Section IV concludes the paper.

II. Methodology

The transition modeling introduced in this paper outperform the dynamic programming approach in computing the expected value, primarily due to their superior efficiency in producing faster results.

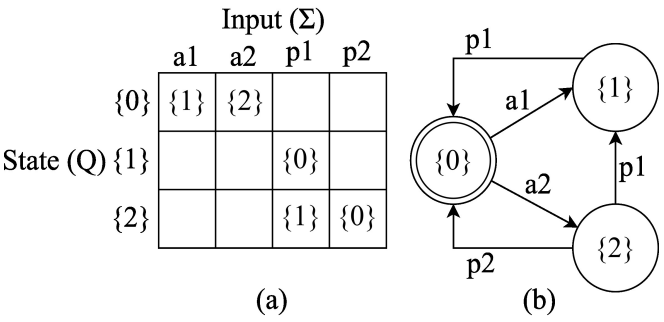

Fig. 3. The finite state machine of the example (cookie crane)

The process for constructing the system to calculate the expected value involves four key steps: mapping the finite state machine, designing a dynamic programming solution to compute the expected value, constructing the matrix representation, and formulating the transition matrix.

A. Finite State Machine Mapping

Mapping the finite state machine is a critical step in identifying the dependencies within the observed object. Each object is associated with a unique finite state machine based on its specific actions and characteristics. For example, while a car can accelerate, a cat can walk, resulting in distinct finite state machines for each. To enable a comprehensive calculation of the property (expected value), the finite state machine is designed to encompass all possible states and scenarios relevant to the context. This ensures that every potential scenario is accounted for [4], [17].

To achieve this, the breadth-first search (BFS) algorithm, a fundamental method for graph traversal, can be implemented. A finite state machine can be represented as a graph, where vertices represent states, and edges represent state transitions. Starting from an initial vertex, BFS systematically traverses the graph by visiting neighboring vertices layer by layer [10]. In this context, each neighboring vertex corresponds to a potential next state, and the traversal action represents a state transition.

In a deterministic finite state machine (FSM), five elements are defined: a finite set of states (Q), a finite, nonempty set of inputs (Σ), a set of transition functions (δ), a starting state (q_0), and a set of accepting states (F). Fig. 3 illustrates these concepts through two representations: (a) a matrix of transition functions (δ), and (b) a graphical depiction of the states and their transitions.

The graphical representation models the behavior of the cookie crane previously discussed. The crane operates in three distinct states, represented as $Q = \{\{0\}, \{1\}, \{2\}\}$, where each state corresponds to a specific configuration or condition of the crane. The state transitions are governed by the set of possible actions, $\Sigma = \{a1, a2, p1, p2\}$. These actions include picking up one cookie ($a1$), picking up two cookies ($a2$), releasing one cookie for packaging ($p1$), and releasing two cookies for packaging ($p2$).

Initially, the cookie crane starts in the empty-handed state, represented as $q_0 = \{0\}$. In this problem, the crane must pick up and package cookies during each iteration, to return to the empty-handed state $\{0\}$, which signifies the completion of the task. Therefore, the finish state is denoted as $F = \{0\}$.

TABLE I
EXAMPLE OF STATE ENCODING FROM STRING TO INTEGER

No.	State	Encoded to
1	{0}	1
2	{1}	2
3	{2}	3

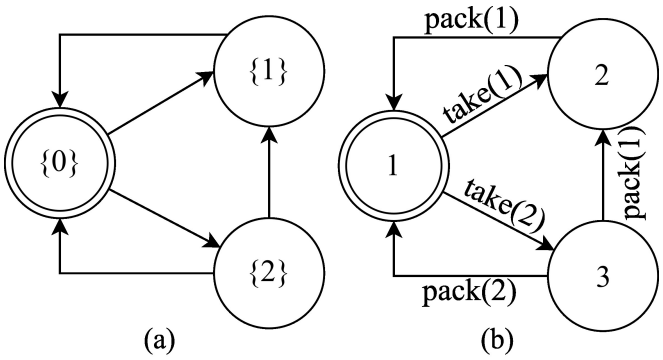

Fig. 4. Encoding state in the finite state machine to integer

To ensure the unique identification of each state, identifiers can be assigned as either numbers or strings. If strings are used, they should be encoded into integers for indexing purposes. This encoding simplifies subsequent mathematical processes. In this problem, the states are encoded into integers, as shown in Table I. It is crucial to map each state to a unique integer identifier to avoid ambiguity and ensure accurate representation. This encoding scheme establishes a one-to-one correspondence between states and their integer identifiers.

As mentioned earlier, string states are encoded into integers. In Fig. 4, the string $\{0\}$ is encoded as integer 1, $\{1\}$ as integer 2, and $\{2\}$ as integer 3. This encoding ensures that each state is uniquely identified by an integer value. Figure (a) shows the finite state machine with string state identifiers, while figure (b) shows the finite state machine with integer state identifiers.

In a single iteration, to produce one package of cookies, the crane must pick up and then release cookies. Let's $take(x)$ represents the action of picking up x cookies and $pack(y)$ represents the action of releasing y cookies. During a single iteration, the crane performs $take(x)$ followed by $pack(y)$ sequentially. Since $take(x)$ and $pack(y)$ are actions that transition the crane between states, they should have the following attributes: the probability of the action occurring (p), the number of cookies picked up or released (c), the starting state (s), and the finishing state (f). For example, $take(x)_p$ represents the probability of picking up x cookies.

To enhance the efficiency of the finite state machine, it is important to merge the $take(x)$ and $pack(y)$ actions into a single action, since both actions occur during a single iteration. The feasibility of merging depends on the context and the object being observed. Fig. 5 illustrates this process: figure (a) shows the original finite state machine, while figure (b) presents the merged version. Merging the actions aims to create a more compact finite state machine, thereby reducing unnecessary further computations.

In Fig. 5, state 1 is the initial state. At the start, state 1 is set as the current state. If $take(1)$ and $pack(1)$ are

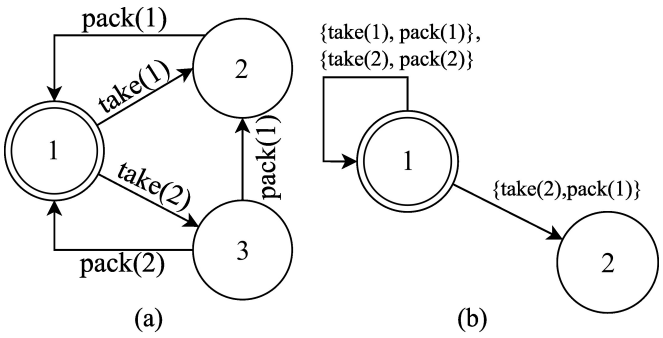

Fig. 5. Merging the state transition in the finite state machine

executed sequentially, the current state will transition from state 1 to state 2 and then back to state 1. Similarly, if $take(2)$ and $pack(2)$ are executed sequentially, the state will change from state 1 to state 3 and then return to state 1. If $take(2)$ and $pack(1)$ are executed sequentially, the state will transition from state 1 to state 3 and then to state 2. Since each iteration consists of one $take(x)$ followed by one $pack(y)$, these actions can be merged into a single action called $prod(z)$. Table II shows all possible combinations of $take(x)$ and $pack(y)$ that form $prod(z)$.

The action $take(x)$ can occur with a certain probability, denoted as $take(x)_p$. Similarly, the action $pack(y)$ has a probability of occurrence denoted as $pack(y)_p$. Since these two actions are independent, the probability of both actions occurring in sequence is the product of their probabilities [1]. Equation (1) illustrates how the probability of $prod(z)_p$ is calculated.

$$prod(z)_p = take(x)_p \times pack(y)_p \tag{1}$$

There may be multiple actions $prod(z)$ with the same starting and finishing states. For example, in Table II, both $prod(1)$ and $prod(2)$ share the same start and finish states. According to the sum rule, if there are two mutually exclusive events, A and B, the probability of either A or B occurring is given by Equation (2) [1].

$$P(A \cup B) = P(A) + P(B) \tag{2}$$

Each $pack(z)$ action is mutually exclusive, representing distinct scenarios that do not depend on each other. Therefore, applying (2) to determine the probability of transitioning from state a to state b is possible. Equation (3) shows how the probability from state a to state b is calculated, denoted as $P_{a,b}$. Let W be the set of $prod(z)$, defined as $W = \{prod(1), prod(2), \ldots, prod(n)\}$.

$$P_{a,b} = \sum_{e}^{\{e \in W | e_s = a \wedge e_f = b\}} e_p \tag{3}$$

Equation (3) states that the probability $P_{a,b}$ of transitioning from state a to state b is determined by summing the probabilities of all elements e in the set W (where e is a $prod(z)$ action) such that $e_s = a$ and $e_f = b$. In this equation, e_p represents the probability associated with action e.

TABLE II
ACTION PROD GENERATED FROM ACTION TAKE AND ACTION PACK

No.	Selected Action Take	Selected Action Pack	Formed Action Prod	State Transition
1	take(1)	pack(1)	prod(1)	1,2,1
2	take(2)	pack(2)	prod(2)	1,3,1
3	take(2)	pack(1)	prod(3)	1,3,2

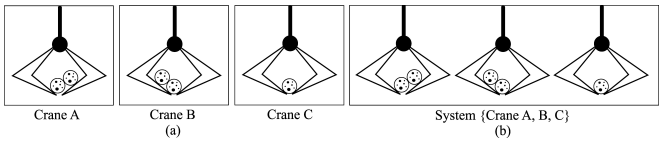

Fig. 6. Collective operation of machines as a single system

Further, according to the expected value formula, it is possible to calculate the expected value by multiplying the probability of a random variable by its corresponding value [1], [17]. In the given problem, the objective is to calculate the expected value of the cookies packaged. Therefore, the random variable represents the number of cookies packaged during the $prod(z)$ action. Equation (4) shows the method for calculating the expected value in this scenario. Let $E_{a,b}$ denote the expected value of cookies packaged when transitioning from state a to state b.

$$E_{a,b} = \sum xf(x) = \sum_{e}^{\{e \in W | e_s = a \wedge e_f = b\}} e_p e_c \qquad (4)$$

Equation (4) defines $E_{a,b}$, the expected value of cookies packaged when transitioning from state a to state b, as the sum of the products of the probability and the number of cookies produced for each element e in the set W (where e is a $prod(z)$ action), such that $e_s = a$ and $e_f = b$. By using this equation, calculating every $E_{a,b}$ for all state transitions can be performed. The expected value is calculated by multiplying each possible scenario with its likelihood to occur and then summing all of the values [17].

The given problem, however, is quite complex. Multiple cookie cranes may elaborate as a system, as illustrated in Fig. 6, thereby increasing the complexity of calculating the expected value due to the combination of involved configuration scenarios.

In Fig. 6, (a) Crane A, B, and C are identical machines, but they may pick up different quantities of cookies. (b) When cranes A, B, and C operate together, they form a unified system. In the given problem, a system can comprise N cookie cranes. At maximum, N can be 4. At a minimum, N can be 1. Therefore, to represent the state of the system, an array notation is utilized. For instance, the state is denoted as $\{c_1, c_2, c_3\}$ for a system consisting of three machines where c_i represents the number of cookies held by the i-th machine at the current state.

In Fig. 7, two distinct conditions exist for a system comprising three cookie cranes. These conditions are represented by state $\{1, 2, 2\}$ and $\{2, 2, 1\}$. Given that the cookie cranes are identical in this problem, both states are considered equivalent as they function within the same system. In other words, the permutation rule does not apply; instead, the combination rule is applicable.

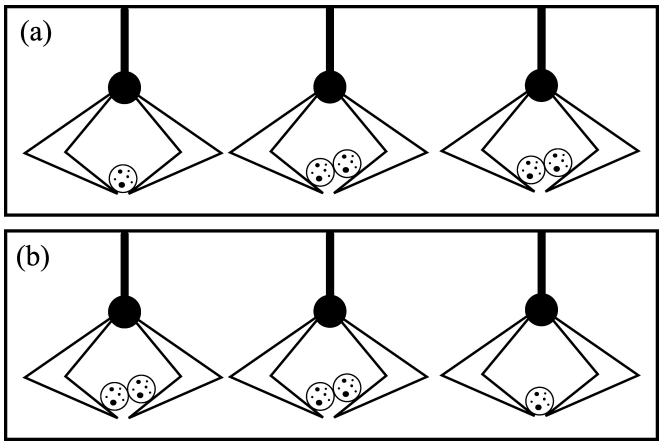

Fig. 7. Two configurations for a system with three machines

TABLE III
CALCULATION OF PROBABILITY OF POSSIBLE SCENARIOS DURING ACTION TAKE OF THE SAMPLE TEST CASE

Initial State	S_{take}	Finish State	Probability (P_{take})
{0,0,0}	{1,1,1}	{1,1,1}	$(3!/(3!1!))(0.5)^3 = 0.125$
{0,0,0}	{2,1,1}	{2,1,1}	$(3!/(1!2!))(0.5)^3 = 0.375$
{0,0,0}	{2,2,1}	{2,2,1}	$(3!/(2!1!))(0.5)^3 = 0.375$
{0,0,0}	{2,2,2}	{2,2,2}	$(3!/3!)(0.5)^3 = 0.125$
{2,0,0}	{0,1,1}	{2,1,1}	$(2!/2!)(0.5)^2 = 0.25$
{2,0,0}	{0,2,1}	{2,2,1}	$(2!/(1!1!))(0.5)^2 = 0.5$
{2,0,0}	{0,2,2}	{2,2,2}	$(2!/2!)(0.5)^2 = 0.25$

TABLE IV
CALCULATION OF PROBABILITY OF POSSIBLE SCENARIOS DURING ACTION PACK OF THE SAMPLE TEST CASE

Start State	S_{pack}	Finish State	Probability (P_{pack})
{1,1,1}	{1,1,1}	{0,0,0}	1.0
{2,1,1}	{2,1,1}	{0,0,0}	1.0
{2,2,1}	{2,2,1}	{0,0,0}	1.0
{2,2,2}	{2,2,0}	{2,0,0}	$3/(3+1) = 0.75$
{2,2,2}	{2,2,2}	{0,0,0}	$1/(3+1) = 0.25$

In the sample test case of the given problem, each cookie crane in the system has a 50% probability of taking either one cookie or two cookies [2]. The desired number of cookies in a package is denoted as G and its value is 5. The system has 3 cookie cranes working together. The probabilities of possible scenarios during the action take and the action pack have been calculated by implementing the product rule and the combination rule. Table III shows the calculation during the action take and Table IV shows the calculation during the action pack.

Table V provides an overview of the process involved in creating the finite state machine according to the sample input data described in Table III and Table IV. It encompasses all possible scenarios of the system, including

TABLE V
POSSIBLE SCENARIOS FROM ACTION TAKE AND ACTION PACK

Action	Initial State	S_{take}	P_{take}	S_{pack}	P_{pack}	Cookies Packaged ($x = \sum S_{pack}$)	End State
prod(1)	{0,0,0}	{1,1,1}	0.125	{1,1,1}	1.0	3	{0,0,0}
prod(2)	{0,0,0}	{2,1,1}	0.375	{2,1,1}	1.0	4	{0,0,0}
prod(3)	{0,0,0}	{2,2,1}	0.375	{2,2,1}	1.0	5	{0,0,0}
prod(4)	{0,0,0}	{2,2,2}	0.125	{2,2,2}	0.25	6	{0,0,0}
prod(5)	{0,0,0}	{2,2,2}	0.125	{2,2,0}	0.75	4	{2,0,0}
prod(6)	{2,0,0}	{0,1,1}	0.25	{2,1,1}	1.0	4	{0,0,0}
prod(7)	{2,0,0}	{0,2,2}	0.25	{2,2,0}	0.75	4	{2,0,0}
prod(8)	{2,0,0}	{0,2,1}	0.5	{2,2,1}	1.0	5	{0,0,0}
prod(9)	{2,0,0}	{0,2,2}	0.25	{2,2,2}	0.25	6	{0,0,0}

the actions taken during $take(i)$ and $pack(j)$ actions. The value of G plays a crucial role in forming Table V. For $prod(1)$, $prod(2)$, and $prod(3)$ actions, they hold fewer cookies than G. Consequently, all the cookies they hold are released during the release or package action. As a result, the state eventually becomes $\{0, 0, 0\}$, indicating that all cranes become empty. However, for $prod(4)$ action, it holds more cookies than G, leading to a hybrid scenario where the cranes can either pack all cookies or pack some cookies only. Fig. 8 provides a visual representation of this scenario.

In scenario $prod(4)$ and $prod(5)$ present different scenarios. After their $take(x)$ actions, they hold 6 cookies. Subsequently, there are two possible outcomes for their $pack(y)$ action: they can release either 4 cookies or 6 cookies. This variability arises because the difference between G and the number of cookies held by the cranes is equal to 1 ($|G - 4| = |G - 6|$) and it is impossible to achieve a combination with a cookie difference of 0.

The system can have multiple outcomes in each scenario, adding complexity to the finite state machine. As shown in Fig. 8, this machine is complex, with some states having multiple transitions. It can be simplified by applying (3) and (4) and encoding state names as unique integers, as explained earlier. In this figure, p represents the probability of the state transition and x represents the number of cookies packaged during the state transition which is the random variable in the given problem. The results of these simplifications are shown in Fig. 9.

The values of $P_{i,j}$ in Fig. 9 are shown by (5), (6), (7), and (8). In these equations, the sum rule has been applied as mentioned in (2). This approach simplifies the finite state machine, ensuring that each state has only one transition to any other state.

$$P_{0,1} = prod(5)_p = (0.125)(0.75) = 0.09375 \tag{6}$$

$$\begin{aligned} P_{1,0} &= prod(6)_p + prod(8)_p + prod(9)_p \\ &= (0.25)(1.0) + (0.5)(1.0) + (0.25)(0.25) \\ &= 0.8125 \end{aligned} \tag{7}$$

$$P_{1,1} = prod(7)_p = (0.25)(0.75) = 0.1875 \tag{8}$$

State 0 can transition to state 1 with a certain probability of $P_{0,1}$. Similarly, state 1 can transition to state 0 with a certain probability of $P_{1,0}$. If the crane transitions from state 0 to state 1 and then from state 1 to state 0 in a sequence, then the probability can be calculated by using the Markov Chain property.

Let X be a random variable, where $P(X = i)$ represents the probability that X takes the value i. The stochastic process $\{X_n : n \in N\}$ is a Markov chain if it satisfies the Markov property, as defined in (9) [5].

$$P(X_{n+1} = j|X_0, X_1, \ldots, X_{n-1}, X_n) = P(X_{n+1}|X_n) \tag{9}$$

In (9), n indicates time, specifically the iteration number. For example, $P(X_0 = i)$ represents the probability that at the first iteration, a movement from state 0 to state i occurs. $P(X_1 = j|X_0)$ denotes the probability such that at the second iteration, a movement from state i to state j occurs, given that a movement from state 0 to state i occurred in the first iteration.

According to the Markov Chain property, a sequence of states visited over time has been considered. Let S denotes the set of visited states. Starting from state 0, the crane sequentially moves to each state in S. In this scenario, the Markov Chain property can be applied to calculate the probability of visiting each state in S in sequence. Equation (10) shows how this probability is calculated.

$$\begin{aligned} P(X_i = S_i \mid X_0, X_1, \ldots, X_{i-1}) = \\ P(X_{i-1} \mid X_{i-2}) P_{S_{i-1}, S_i} \end{aligned} \tag{10}$$

B. DP Solution to Compute Expected Value

When calculating the expected value, the outcomes depend on the state transition probabilities mapped within the finite state machine. Each transition from one state to another encompasses multiple scenarios that occur with specific probabilities. According to the mathematical expectation formula [1], the expected value is calculated as shown in (11).

$$E(X) = \sum_x x f(x) \tag{11}$$

To calculate the expected value for the problem of SPOJ TAP2015E, equation (11) is applied. In a single iteration, there are no dependencies because there is only one state transition. However, for multiple iterations, dependencies between states arise. In this problem, multiple iterations (where 1 iteration visits 1 state) can occur. Let M represents the number of iterations. For $M > 1$, the probability is calculated using (10).

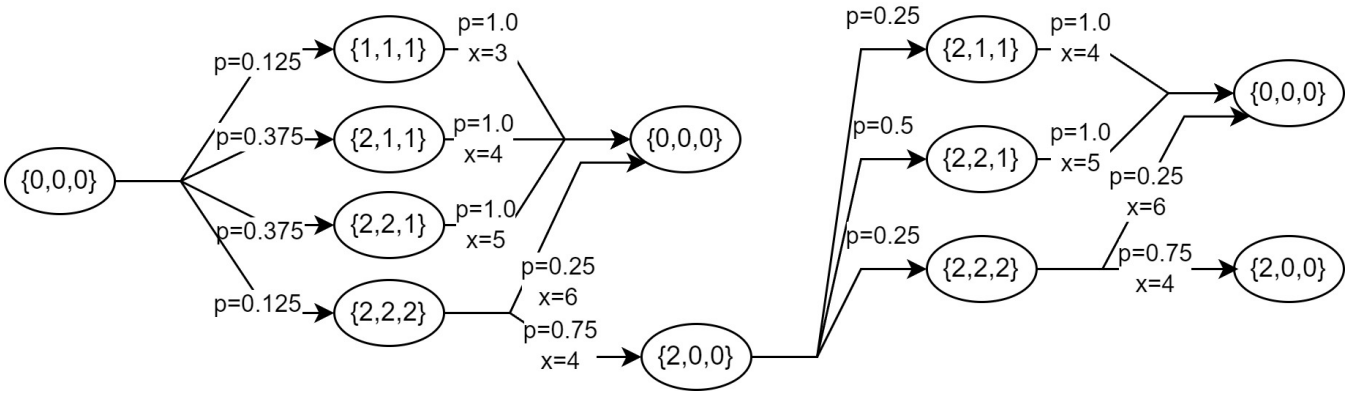

Fig. 8. Possible scenarios of machines starting empty and then each taking a different number of cookies

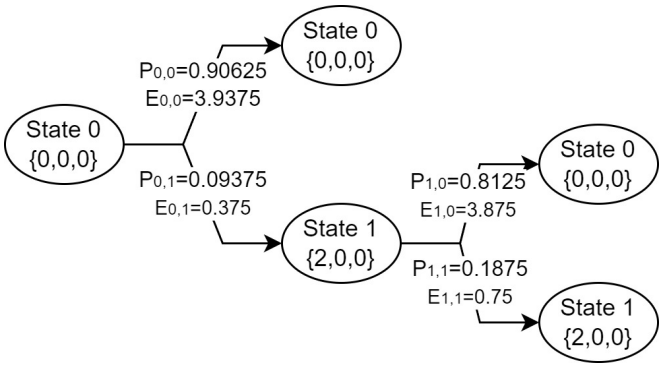

Fig. 9. The finite state machine with encoded states and simplified transitions

The Law of Total Expectation has been applied, as shown in (12). According to the Law of Total Expectation, the expected value of a random variable equals the sum of the expected values conditioned on another random variable [6]. By applying the Law of Total Expectation, the expected value can be calculated based on the iteration number, which corresponds to the number of movements.

$$E(X) = E(E(X|A)) = \sum_{i=1}^{n} E(X|A_i)P(A_i) \tag{12}$$

In the first iteration ($m = 1$), as mentioned, there is no dependency because it only has one state transition. Hence, $m = 1$ can be served as the terminating case. For convenience, a notation $E(X|M = m, Q = q)$ is introduced. In (13), m represents the iteration number and q represents the current state during that m-th iteration. This notation denotes the expected value of the random variable X during the m-th iteration with the current state q. In (13), n represents the highest integer in the set of encoded states.

$$E(X|M = 1, Q = q) = \sum_{j=0}^{n} E_{0,j} \tag{13}$$

For $m > 1$, dependencies exist because the current state in the x-th iteration depends on the finish state in the $(x-1)$-th iteration. If the crane transitions from state A to state B in the 1st iteration, then state B becomes the start state in the 2nd iteration. Based on (12), (14) shows the application of the Law of Total Expectation in this problem. With (13) and (14), a recurrence function can be established as shown in (15).

The objective of (15) is to compute the expected value of the random variable X up to the depth of m in the iteration. Consequently, the mean value of these expected values can be calculated ranging from $m = 1$ to $m = M$. The final output is determined using (16). Equation (16) aggregates the expected values $E(X|M = m, Q = q)$ across all iterations from 1 to M, to provide the overall expected value of X after M iterations.

$$E(X) = \frac{1}{M} \sum_{m=1}^{M} E(X|M = m, Q = 0) \tag{16}$$

Based on (15) and (16), the dynamic programming solution for this problem is presented in algorithm in Fig 10. This algorithm utilizes tabulation, an optimization technique that enhances efficiency by precomputing solutions to subproblems and storing them in a table [18]. By leveraging tabulation, algorithm in Fig. 10 efficiently computes the expected value of the random variable X after M iterations, ensuring faster execution.

In the algorithm presented in Fig. 10, P denotes a matrix that stores the probabilities of state transitions, while E represents a matrix containing the expected values associated with these transitions. The parameter M specifies the iteration count, which corresponds to the number of state transitions performed. Further details about these matrices are provided in Section II.C.

C. Matrix Representation

As previously mentioned, matrices play a crucial role in storing both the probabilities and the expected values of state transitions. Each state transition yields an associated expected value. Accordingly, matrix P is defined to represent transition probabilities, while matrix E is defined to represent the expected values, as expressed in (17) and (18).

$$P = \begin{bmatrix} P_{0,0} & P_{0,1} & \dots & P_{0,n} \\ P_{1,0} & P_{1,1} & \dots & P_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n-1,0} & \dots & \dots & P_{n-1,n} \end{bmatrix} \tag{17}$$

$$\begin{aligned} P_{0,0} &= prod(1)_p + prod(2)_p + prod(3)_p + prod(4)_p \\ &= (0.125)(1.0) + (0.375)(1.0) + (0.375)(1.0) + (0.125)(0.25) \\ &= 0.90625 \end{aligned} \tag{5}$$

$$E(X|M = m, Q = q) = \sum_{j=0}^{n} E(X|M = m-1, Q = j)P_{q,j} \tag{14}$$

$$E(X|M = m, Q = q) = \begin{cases} \sum_{j=0}^{n} E_{0,j} & , m = 1 \\ \sum_{j=0}^{n} E(X|M = m-1, Q = j)P_{q,j} & , m > 1 \end{cases} \tag{15}$$

Require: P, E, M
Ensure: $answer$
1: **for** $q \leftarrow 0$ to $n-1$ **do**
2: $dp[0][q] \leftarrow 0.0$
3: **for** $v \leftarrow 0$ to $n-1$ **do**
4: $dp[0][q] \leftarrow dp[0][q] + E[q][v]$
5: **end for**
6: **end for**
7: $sum \leftarrow dp[0][0]$
8: **for** $m \leftarrow 1$ to $M-1$ **do**
9: **for** $q \leftarrow 0$ to $n-1$ **do**
10: $dp[m \bmod 2][q] \leftarrow 0.0$
11: **for** $v \leftarrow 0$ to $n-1$ **do**
12: **if** $P[q][v] > 0.0$ **then**
13: $dp[m \bmod 2][q] \leftarrow dp[m \bmod 2][q] + dp[(m-1) \bmod 2][v] \times P[q][v]$
14: **end if**
15: **end for**
16: **end for**
17: $sum \leftarrow sum + dp[m \bmod 2][0]$
18: **end for**
19: $answer \leftarrow sum/M$

Fig. 10. Dynamic programming algorithm solution for the problem

$$E = \begin{bmatrix} E_{0,0} & E_{0,1} & \dots & E_{0,n} \\ E_{1,0} & E_{1,1} & \dots & E_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ E_{n-1,0} & \dots & \dots & E_{n-1,n} \end{bmatrix} \tag{18}$$

P and E are square matrices, meaning they have an equal number of rows and columns [7], [18]. The element $P_{i,j}$ represents the probability of transitioning from state i to state j, while $E_{i,j}$ denotes the expected value generated during the transition from state i to state j. These matrices are capable of representing fully connected state transitions, where any state can transition to any other state. Consequently, the size of these matrices, denoted as n, corresponds to the total number of states in the finite state machine.

In C++ programming, matrices are commonly represented using 2-dimensional arrays [9] due to their straightforward structure, which closely mirrors the mathematical representation of matrices. Both matrix addition and matrix multiplication require iterative computations over the elements of the matrices. The algorithm in Fig. 11 illustrates how

Require: A, B
Ensure: C
1: $n \leftarrow A.rowSize$
2: $m \leftarrow A.colSize$
3: **for** $i \leftarrow 0$ to $n-1$ **do**
4: **for** $j \leftarrow 0$ to $n-1$ **do**
5: $C[i][j] \leftarrow A[i][j] + B[i][j]$
6: **end for**
7: **end for**

Fig. 11. Matrix addition algorithm

Require: A, B
Ensure: C
1: $r1 \leftarrow A.rowSize$
2: $c2 \leftarrow B.colSize$
3: $c1 \leftarrow A.colSize$
4: **for** $i \leftarrow 0$ to $r1-1$ **do**
5: **for** $j \leftarrow 0$ to $c2-1$ **do**
6: **for** $k \leftarrow 0$ to $c1-1$ **do**
7: $C[i][j] \leftarrow C[i][j] + A[i][k] + B[k][j]$
8: **end for**
9: **end for**
10: **end for**

Fig. 12. Naïve matrix multiplication algorithm

matrix addition is performed, while Fig. 12 presents a naïve iterative approach to matrix multiplication. A more efficient iterative method for matrix multiplication is discussed in the subsequent sections.

The algorithm in Fig. 11 has a time complexity of $O(N^2)$, which arises from its two nested loops, each iterating over the rows and columns of the matrix. In contrast, the algorithm in Fig. 12 has a time complexity of $O(N^3)$ because it involves three nested loops: two for iterating over the rows and columns, and one for summing over the elements during matrix multiplication. In both algorithms, the methods '$A.rowSize$' and '$A.colSize$' return integers representing the number of rows and columns in matrix A, respectively.

A computer's memory is a collection of cells, each of which stores one bit of information [8]. The computer's memory (primary storage) has a linear structure [9]. Typically, memory addresses range from 0 to an upper limit determined by the amount of memory available on the given computer. Consider there is an array of integer, $A[4][5]$, i.e.,

Require: A, Bt
Ensure: C
1: $r1 \leftarrow A.rowSize$
2: $c2 \leftarrow Bt.rowSize$
3: $c1 \leftarrow A.colSize$
4: **for** $i \leftarrow 0$ to $r1 - 1$ **do**
5: **for** $j \leftarrow 0$ to $c2 - 1$ **do**
6: **for** $k \leftarrow 0$ to $c1 - 1$ **do**
7: $C[i][j] \leftarrow C[i][j] + A[i][k] + B[j][k]$
8: **end for**
9: **end for**
10: **end for**

Fig. 13. Matrix multiplication algorithm with matrix transpose

an array of integers named A having 4 rows and 5 columns. The address of $A[0][0]$ is called the base address, BA. Suppose each element of the integer occupies 4 bytes. $A[0][0]$ is stored at memory location BA. $A[0][1]$ is stored at memory location $BA+4$. $A[0][2]$ is stored at memory location $BA+8$. $A[0][3]$ is stored at memory location $BA + 12$. $A[0][4]$ is stored at memory location $BA + 16$. $A[1][0]$ is stored at memory location $BA + 20$. $A[3][4]$ is stored at memory location $BA + 76$.

During an execution of a program, memory references by a processor, for both instructions and data. The processor cache contains a copy of a portion of the main memory. When the processor attempts to read a byte of memory, a check is made to determine whether the byte is in the cache. If so, the byte is delivered to the processor immediately. If not, a block of memory which consists of a fixed number of bytes is read into the cache. Further, the byte is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that many of the near-future memory references will be to other bytes in the block [11].

As mentioned above, when the processor accesses an element of the matrix in memory, it checks whether the element is already present in the cache. If the element is not in the cache, a block of memory containing several elements of the matrix is loaded into the cache. In the algorithm shown in Fig. 12, the elements of matrix B are not accessed in a sequential memory address order. As a result, data is loaded from memory to the cache more frequently. Furthermore, when new data is written to the cache, it may cause existing cached data to be flushed to make room for the new block. This repeated loading and flushing of data increases latency and reduces computational efficiency.

To accelerate matrix multiplication, matrix transposition has been incorporated into the algorithm shown in Fig. 12. The optimized approach is presented in Fig. 13, which demonstrates a more efficient method for matrix multiplication. By transposing one of the matrices, the algorithm improves memory access patterns, reducing repetitive cache loads and increasing the cache hit rate [19]. This optimization minimizes latency caused by frequent memory-to-cache transfers, thereby enhancing overall computational performance.

In the algorithm shown in Fig. 13, Bt represents the transpose of B, denoted as $Bt = B^T$. The access sequence for the array indices follows the order (i, j), (i, k), and (j, k). This sequence ensures that the elements of the arrays are accessed in ascending order of memory addresses. For instance, when computing $C[1, 2]$, the processor cache is likely to already contain the data for $A[1, x]$ and $Bt[2, x]$ for $x = \{1, 2, 3, \ldots, c1\}$. This is because the data for $A[1, x]$ and $Bt[2, x]$ was loaded into the cache during the calculation of the previous element, $C[1, 1]$. By leveraging this memory access pattern, the approach optimizes cache usage and significantly accelerates matrix multiplication.

D. Transition Matrix

The transition matrix model can vary significantly depending on the context and specific rules of the problem. For the SPOJ TAP2015E problem, a final matrix, denoted as F, has been constructed to address its unique requirements. This matrix is used to solve (16) through matrix transitions. Equation (19) illustrates how matrix F substitutes the variables in (16), reflecting the specific context of this scenario.

$$E(X) = \frac{1}{M} \sum_{j=0}^{n} F(M)_{0,j} \tag{19}$$

In (19), $F(M)$ represents the final matrix for the M-th iteration. For example, $F(3)$ is the final matrix for the 3rd iteration. It is important to note that $F(x) \neq F(y)$ for $x \neq y$ and $1 \leq x < y \leq M$. Since $F(M)$ is a matrix, its element can be accessed by using the notation $F(M)_{i,j}$ for the i-th row and j-th column. For instance, $F(M)_{0,j}$ refers to the element in the first row and j-th column of the matrix. For the terminator case or base case where $M = 1$, the matrix F has been defined as (20).

For $M = 2$, the Law of Total Expectation is applied, indicating that there is a dependency between the expected value at $M = 2$ and the expected value at $M = 1$. This dependency must be considered to accurately calculate the expected value for $M = 2$. By using matrix multiplication, based on (15), $F(M)$ has been defined as (21).

$$F(M) = E + PE + P^2E + \ldots + P^{M-1}E \tag{21}$$

In the properties of matrix arithmetic, there is distributive law $A(B+C) = AB+AC$ [7]. The reversed version of this distributive law has been applied to matrix multiplication in (21), yielding the result shown in (22).

$$F(M) = (I + P + P^2 + \ldots + P^{M-1})E \tag{22}$$

In (22), a geometric series of matrices can be observed. The sequence $\{S_n\}_{n\geq 0}$ is defined by (23). $\{S_n\}_{n\geq 0}$ is the geometric series generated by P. The series converges if and only if $|\lambda_i| < 1$, for each eigenvalue λ_i of P [12]. If this condition holds, then $I - P$ is invertible and results in (24).

$$S_n = I + P + P^2 + \ldots + P^{n-1}, S_0 = I \tag{23}$$

$$S_n = \sum_{k=0}^{n-1} P^k = (I - P)^{-1}(I - P^n) \tag{24}$$

If the geometric series does not converge, (24) cannot be applied for computation. In the given problem, the geometric

$$F(1) = \begin{bmatrix} E(X|M=1,Q=0) & E(X|M=2,Q=0) & \dots & E(X|M=n,Q=0) \\ E(X|M=1,Q=1) & E(X|M=2,Q=1) & \dots & E(X|M=n,Q=1) \\ \vdots & \vdots & \ddots & \vdots \\ E(X|M=1,Q=n-1) & \dots & \dots & E(X|M=n,Q=n-1) \end{bmatrix} \quad (20)$$

series generated by P does not converge. Therefore, (24) cannot be utilized in the given transition matrix model. Instead, our proposed algorithm can be used with logarithmic time complexity to efficiently compute the series, known as the Sum of Powered Matrix algorithm. Let $B(x)$ denotes the function used to compute the geometric series generated by P starting from index 1. This function is defined in (25).

$$B(x) = P + P^2 + \ldots + P^x = \sum_{u=1}^{x} P^u \quad (25)$$

If $x = 1$, then $B(1) = P$. For $x = 2$, $B(2) = P + P^2$. Similarly, for $x = 3$, $B(3) = P + P^2 + P^3$, and so forth. This function is computed sequentially, meaning $B(x)$ is calculated before $B(x+1)$ for $x > 1$. However, this sequential approach is not computationally efficient for larger values of x. For example, if $M = 10^7$, calculating $B(10^7)$ directly using this method would be computationally expensive and time-consuming, particularly in a worst-case scenario. Therefore, a more optimized approach is necessary to handle such cases efficiently.

To accelerate the computation of $B(x)$ as defined in (25), a logarithmic approach has been proposed. Instead of sequentially calculating $B(x)$ from $B(1)$ to $B(M-1)$, the proposed method constructs $B(x)$ using logarithmic steps. This approach optimizes the computation process by employing efficient iterative calculations, enabling significantly faster computation, even for large values of x, such as $M = 10^7$.

For $B(2)$, the reversed version of the distributive law of matrices is applied, as shown in (26). Similarly, for $B(3)$, the reversed version of the distributive law of matrices is applied, followed by substituting it with (26). This process continues for $B(4)$, where the reversed version of the distributive law of matrices is applied and then substituted using (26). Consequently, by utilizing (26), (27), and (28), $B(x)$ can be computed in approximately $\log_2 x$ steps. Equation (29) provides the formula to compute $B(x)$ efficiently using logarithmic steps.

$$B(2) = P + P^2 = P(I + P) \quad (26)$$

$$B(3) = P + P^2 + P^3 = P(I+P) + P^3 = B(2) + P^3 \quad (27)$$

$$\begin{aligned} B(4) &= P + P^2 + P^3 + P^4 \\ &= (P + P^2)(I + P^2) \\ &= P(I + P)(I + P^2) \\ &= B(2)(I + P^2) \end{aligned} \quad (28)$$

$$B(x) = \begin{cases} P & , x = 1 \\ B(\lfloor \frac{x}{2} \rfloor)(I + P^{\lfloor \frac{x}{2} \rfloor}) & , x \bmod 2 = 0 \\ B(\lfloor \frac{x}{2} \rfloor)(I + P^{\lfloor \frac{x}{2} \rfloor}) + P^x & \end{cases} \quad (29)$$

Require: x
Ensure: $B(x) = T$

```
factor ← vector()
while x > 0 do
    factor.push_back(x)
    x ← x shr 1
end while
xBef ← 0
for i ← factor.size − 1 downto 0 do
    f ← factor[i]
    if f = 1 then
        T ← P
    else
        halfF ← f shr 1
        if halfF = 1 then
            X ← P
            xBef ← 1
        else
            X ← X × X
            xBef ← xBef × 2
            if xBef ≠ halfF then
                xBef ← xBef + 1
                X ← X × P
            end if
        end if
        Y ← I + X
        T ← T × Y
        if f mod 2 = 1 then
            Z ← X × X
            if xBef × 2 ≠ f then
                Z ← Z × P
            end if
            T ← T + Z
        end if
    end if
end for
```

Fig. 14. Sum of powered matrix algorithm

The algorithm presented in Fig. 14 introduces the Sum of Powered Matrix, a novel approach developed in this paper as a key performance-enhancing technique. Designed to solve (29), this algorithm is implemented using a bottom-up approach, enabling efficient computation of $B(x)$ in a logarithmic number of steps relative to x. By significantly reducing computational overhead, this method ensures optimal performance, even for large values of x, and represents a cornerstone of the performance improvements achieved in this work.

In the algorithm shown in Fig. 14, the process begins by storing the value of x in a vector. Subsequently, x is repeatedly divided by 2 until it becomes zero. This process continues until the vector's size reaches $\log_2 x$,

which corresponds to the number of iterations performed by the algorithm. As a result, the algorithm achieves a time complexity of $O(\log_2 N)$.

By applying (29), the transition matrix model is ultimately formulated in (30). This formulation encapsulates the iterative accumulation of matrix powers, leveraging the efficient Sum of Powered Matrix algorithm. This approach ensures robust and accurate computation of $B(x)$ with logarithmic complexity, highlighting its efficiency.

$$F(M) = \begin{cases} E & , M = 1 \\ (I + B(M-1))E & , M > 1 \end{cases} \tag{30}$$

Once matrix F is constructed, the expected value is calculated using (19) and (30). To compute $F(M)$ efficiently, the matrix multiplication in the equation is performed using the transpose matrix multiplication technique. This approach optimizes cache management by improving memory access patterns, reducing cache misses, and significantly accelerating the computation process. As a result, the expected value is calculated efficiently, even for large values of M.

E. Converting the Final Equation into Transition Matrix Model

Equation (19) represents the matrix transition approach for solving the given problem, while Equation (16) represents the recurrence function approach. Both equations yield the same result; however, the key distinction lies in their methodology. The first equation incorporates a matrix (F) as part of the solution, whereas the second equation does not involve any matrices.

$$\begin{aligned} E(X) &= E(X) \\ \frac{1}{M}\sum_{j=0}^{n} F_{0,j} &= \frac{1}{M}\sum_{m=1}^{M} E(X|M=m, Q=0) \\ \sum_{j=0}^{n} E_{0,j} &= E(X|M=1, Q=0) \\ \sum_{j=0}^{n} E_{0,j} &= \sum_{j=0}^{n} E_{0,j} \end{aligned} \tag{31}$$

For $M = 1$, (31) shows that (16) is equivalent to the sum of the elements in matrix F. Additionally, for $M > 1$, (32) demonstrates that (16) is derived from the matrix multiplication of matrices P and E, resulting in matrix F. This highlights that converting a linear approach into a transition matrix approach relies on the specific formulation of the linear equation. In this process, multiple matrices are employed to represent the elements of the linear equation. Due to their interdependence, these elements can be computed efficiently and simultaneously, leveraging the power of matrix operations.

III. Experimental Results and Analysis

In the previous section, how the approach of the expectation value problem using this finite state machine and matrix transition has been explained. All proposed algorithms in this paper were implemented using C++ programming language. In this section, the proposed algorithm will be examined by using two different platforms suitable for analyzing its validity and performance.

The first platform is the Sphere Online Judge (SPOJ). It is a third-party online platform for source code checking. It uses cube clusters with Intel Xeon E3-1220 v5 CPUs [13]. The validity of the matrix transition algorithm was tested by submitting the source code on SPOJ. It examined the submitted source code validity by comparing its result output to the expected answer provided by the problem originator.

The second platform is a local environment using a personal computer (PC) with AMD Ryzen™ 5 2500U and 8192MB of memory, running Windows 10 with GCC 4.9.2 compiler. This local environment is used to compare and analyze the program's runtime and efficiency.

In Section III.A, the validity check of the matrix transition algorithm that is used to compute the expected value will be elaborated. Moreover, in Section III-B, the performance examination of the proposed method, both time-wise and space-wise will be further going into detail. Lastly, in Section III-C, how efficiently the matrix transition method works in different input sizes will be analyzed.

A. Validity Examination

Fig. 16 shows how the matrix transition approach scored on Sphere Online Judge (SPOJ). SPOJ's testing system will give various response statuses based on its judgment of the submitted solution. "Accepted" status indicates that the program runs successfully and gives a correct answer. "Wrong answer" status means that the program runs successfully but it gives different answers than the expected answer. "Time limit exceeded" shows that the program is compiled successfully but it runs exceeding the time limit. "Compilation error" means that the program is not able to be compiled. Lastly, the "Runtime error" status implies that the program is compiled successfully but it crashed during its runtime [14].

The code was tested across 30 submissions to ensure both accuracy and consistency throughout the validity test. Each of the 30 submissions received an "accepted" status, demonstrating that our approach to computing the expected value using finite state machines and matrix transitions provides correct results within the given time and memory constraints. Each submission was evaluated against multiple hidden test cases configured by the problem originator. The "accepted" status is awarded only when the code passes all test cases, highlighting the robustness of our solution. Furthermore, Fig. 15 illustrates that only 7 out of 172 submitted solutions were accepted by Sphere Online Judge, with our matrix transition solution being one of them. This not only underscores the complexity of the problem but also the effectiveness of our proposed solution.

B. Performance Examination

Two factors must be taken into consideration in the performance examination. They are program runtime and memory usage [15]. The first factor, program runtime, will be evaluated in two distinct environments: locally, utilizing one's personal computer, and in a live environment, facilitated by the Sphere Online Judge platform. This comprehensive approach enables a thorough examination of the program's

$$
\begin{aligned}
E(X|M=2,Q=0) &= \sum_{j=0}^{n} E(X|M=1,Q=j)P_{q,j} \\
&= E(X|M=1,Q=0)P_{0,0} + E(X|M=1,Q=1)P_{0,1} + \ldots + E(X|M=1,Q=n)P_{0,n} \\
&= (\sum_{j=0}^{n} E_{0,j})P_{0,0} + (\sum_{j=0}^{n} E_{1,j})P_{0,1} + \ldots + (\sum_{j=0}^{n} E_{n,j})P_{0,n} \\
&= (E_{0,0} + E_{0,1} + \ldots + E_{0,n})P_{0,0} + (E_{1,0} + E_{1,1} + \ldots + E_{1,n})P_{0,1} + \ldots + \\
&\quad (E_{n,0} + E_{n,1} + \ldots + E_{n,n})P_{0,n} \\
&= P_{0,0}E_{0,0} + P_{0,0}E_{0,1} + \ldots + P_{0,0}E_{0,n} + P_{0,1}E_{1,0} + P_{0,1}E_{1,1} + \ldots + \\
&\quad P_{0,1}E_{1,n} + P_{0,n}E_{n,0} + P_{0,n}E_{n,1} + \ldots + P_{0,n}E_{n,n}
\end{aligned} \tag{32}
$$

Perfect packing statistics & best solutions

Users accepted	Submissions	Accepted	Wrong Answer	Compile Error	Runtime Error	Time Limit Exceeded
7	172	86	38	12	2	33

RANK	DATE	USER	RESULT	TIME	MEM	LANG
1	2022-11-23 08:48:52	Ryan Garnet Andrianto	accepted	0.17	6.4M	CPP
2	2022-12-03 22:40:33	Rully Soelaiman	accepted	0.17	5.8M	CPP
3	2020-05-19 23:29:53	Karolis Kusas	accepted	1.15	4.5M	CPP14
4	2016-06-03 11:56:46	[Rampage] Blue.Mary	accepted	1.22	3.1M	C++ 4.3.2
5	2015-11-05 08:34:43	Alex Anderson	accepted	1.93	1341M	JAVA
6	2016-09-20 19:43:07	Matías Hunicken	accepted	1.94	3.3M	C++ 4.3.2
7	2019-09-25 02:26:09	Jakub Byczkowski	accepted	2.46	4.3M	C++ 4.3.2

Fig. 15. Statistics of all previously submitted solutions and list of accepted users examined by Sphere Online Judge

ID	DATE	USER	RESULT	TIME	MEM	LANG
30534524	2022-12-07 05:49:40	Ryan Garnet Andrianto	accepted	0.18	6.0M	CPP
30534523	2022-12-07 05:49:19	Ryan Garnet Andrianto	accepted	0.18	5.4M	CPP
30534522	2022-12-07 05:49:11	Ryan Garnet Andrianto	accepted	0.17	5.7M	CPP
30534521	2022-12-07 05:49:01	Ryan Garnet Andrianto	accepted	0.17	5.2M	CPP
30534520	2022-12-07 05:48:52	Ryan Garnet Andrianto	accepted	0.17	5.8M	CPP

Fig. 16. Validity test evaluated by Sphere Online Judge

performance across different settings. The second factor, i.e., memory usage will be examined. It will be evaluated exclusively on the Sphere Online Judge site. By focusing on this specific platform, how the program manages and utilizes memory resources can be better understood. This approach allows us to assess the efficiency and effectiveness of the program's memory management in a controlled and standardized environment.

When evaluating the performance of an algorithm, it is essential to incorporate worst-case scenarios into the evaluation process. This approach allows us to ascertain whether the algorithm remains robust and effective even under adverse conditions. By subjecting the algorithm to unfavorable scenarios, its ability to function optimally can be gauged and determine its acceptability in real-world applications. Therefore, to evaluate the program runtime, 9 types of test cases were used. Each test case has different input M which are 1,10,10^2,10^3,10^4,10^5,10^6,5×10^6, and 10^7. Input M denotes how many state transitions. The algorithm should calculate the expected value after M state transitions. In our matrix transition model, the number of iterations affects the computational power. The more it has iteration, the more it has computational tasks because it is in the form of a recursive function.

Fig. 18 shows the scatter chart of every thirty trials' average runtime value for each different input M. It is shown that the program runtime is logarithmic increasing with the M. The data were obtained by executing the implemented approach on the PC 30 times for input shown in Fig. 17.

In the live environment, the algorithm is submitted thirty times to check on its performance consistency. Fig. 19 shows

```
Input:
3 6 5 M
3 0.16
4 0.16
5 0.16
6 0.16
7 0.16
8 0.20
```

Fig. 17. Input for local testing on the PC

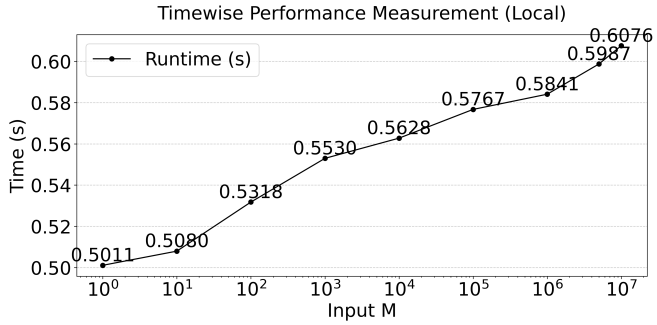

Fig. 18. Average program runtime in different input M

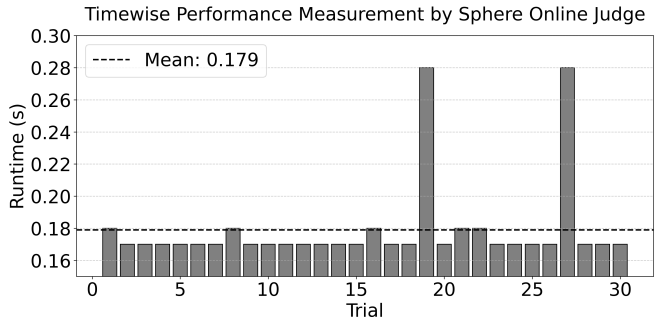

Fig. 19. Timewise performance measurement by Sphere Online Judge

a bar chart of all thirty submission execution times measured by the Sphere Online Judge, with an average execution time of only 0.179 seconds out of 4 seconds, the maximum given time limit. Only two trials (19th and 27th) took 0.28 seconds which is ±100 milliseconds longer than the average runtime. Therefore, these 19th and 27th trials can be considered outliers caused by the server load inconsistency during a certain busy time [16].

As mentioned at the beginning of this section, the proposed approach to memory usage examination is done by the Sphere Online Judge. It shows how many resources were used when executing the submitted program. On average, our proposed approach only needs 5.63MB of resources which is only 0.36% of the memory limit (1536MB). Fig. 20. shows the memory usage on each of the trials. In conclusion, our matrix transition approach is indeed efficient space-wise.

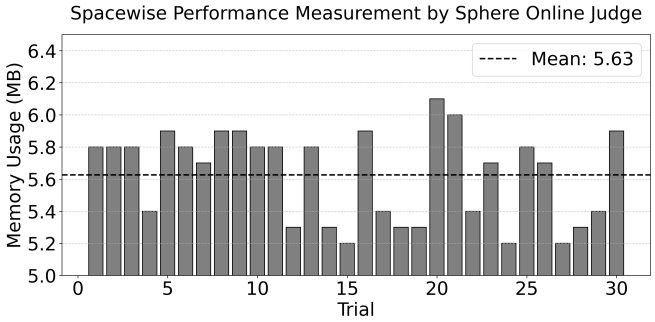

Fig. 20. Space-wise performance measurement by Sphere Online Judge

C. Matrix Transition Method Examination

In this section, the efficiency of our matrix transition algorithm in solving the finite state machine in different numbers of iterations (M) will be analyzed. Our matrix transition algorithm uses bottom-up loops to compute the expected value in each iteration. The more iterations it needs, the longer the runtime will be.

Calculating the expected value using only the dynamic programming (DP) solution involves $n + M \times n^2$ iterations, where n represents the number of states. The complexity of the DP solution, as described in Equation (33), is derived from the total number of iterations required for the computation. Here, z_{dp} denotes the approximate number of iterations, as implemented in the algorithm presented in Fig. 10.

$$z_{dp} = n + M \times n^2 \tag{33}$$

Calculating the expected value using the matrix transition solution requires fewer iterations compared to the dynamic programming solution. To compute the final matrix $F(M)$, the process involves a single matrix addition operation and one matrix multiplication operation, as shown in (30). The matrix addition requires n^2 iterations, where n is the size of the matrix, while the matrix multiplication requires n^3 iterations.

Additionally, the computation of $F(M)$ includes the calculation of $B(M-1)$. This calculation involves $\log_2(M-1)$ iterations. During each of these iterations, as detailed in the algorithm presented in Fig. 14, the worst-case scenario requires performing two matrix addition operations and six matrix multiplication operations. Thus, the total number of iterations for calculating $B(M-1)$ is given by $(\log_2(M-1))(2n^2 + 6n^3)$. Consequently, the total number of iterations to compute $F(M)$ is $n^2 + n^3 + (\log_2(M-1))(2n^2 + 6n^3)$.

Furthermore, deriving the expected value also involves solving (13), which requires M iterations. Therefore, the computational complexity of the matrix transition solution is expressed in (34), where z_{tr} denotes the approximate number of iterations required to solve the problem using matrix transition.

$$z_{tr} = n^2 + n^3 + (\log_2(M-1))(2n^2 + 6n^3) + M \tag{34}$$

In the given problem, based on the input constraints, the maximum number of states is 210. Therefore, $n = 210$ is used to represent the worst-case scenario input. Equation (35)

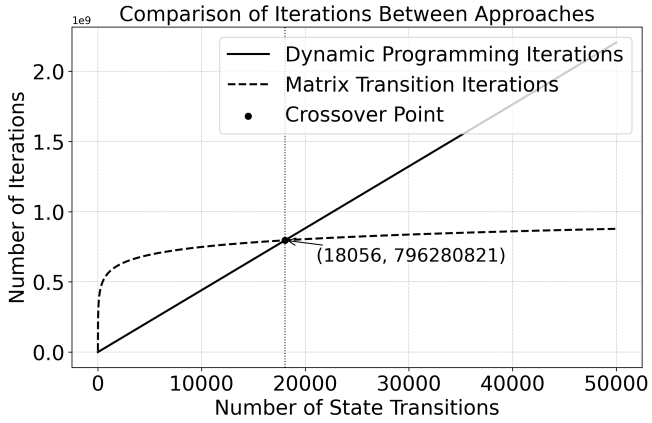

Fig. 21. Comparison of 2D plot complexity between the matrix transition solution and the dynamic programming solution

and (36) illustrate the result of substituting $n = 210$ into equations (33) and (34), respectively.

$$z_{dp} = 210 + M \times 44100 \tag{35}$$

$$z_{tr} = 9305100 + 55654200(\log_2 (M - 1)) + M \tag{36}$$

Subsequently, (35) and (36) are plotted in a 2D graph, where the X-axis represents the value of M, and the Y-axis represents the value of z_{tr} and z_{dp}. Fig. 21 presents this 2D visualization. In this context, a lower value of z indicates improved performance, as z directly corresponds to the number of computational iterations required. The matrix transition approach demonstrates consistently lower iteration counts compared to the dynamic programming (DP) approach, highlighting its computational advantage for solving this problem.

The plot also identifies a *crossover point*, representing the value of M at which both methods exhibit identical computational performance in terms of iterations required. It is important to note that this crossover point is derived through mathematical observation and may slightly differ in practical implementations due to factors such as system-specific overheads or execution nuances. Beyond this threshold, the matrix transition method significantly outperforms the DP approach, requiring fewer computations as the value of M increases. This further underscores the scalability and efficiency of the matrix transition approach for larger state spaces.

The matrix transition method achieves this advantage by leveraging the logarithmic nature of its computations, effectively reducing the complexity compared to the linear growth of the DP approach. As M increases, the iterative multiplications required in the DP approach grow linearly, while the matrix method confines the required operations to approximately $\log_2 M$ steps. This difference becomes increasingly pronounced for larger M, making the matrix transition method more suitable for applications involving large-scale state spaces or computational constraints.

IV. Conclusion

This paper addresses the problem of calculating the expected value of a random variable associated with an object whose states exhibit interdependence. These states are modeled using a finite state machine, where each state is assigned specific value properties essential for accurate computation. To enhance computational efficiency and scalability, a matrix representation is utilized. The expected value is determined through matrix transitions, supported by an algorithm with logarithmic complexity that accelerates the computation of geometric series of matrices, resulting in significant performance improvements.

Experimental results demonstrate that the proposed approach, combining finite state machines with matrix transitions, consistently produces accurate results while optimizing computational resources, including time and memory. The method achieves exceptional efficiency, making it well-suited for large-scale or resource-constrained scenarios. Future work should focus on enhancing this approach to tackle more complex problems and expanding its applicability to diverse domains, addressing the growing demand for efficient and scalable computational techniques.

References

[1] E. W. Ronald, H. M. Raymond, L. M. Sharon, and K. Ye, "Probability & Statistics for Engineers & Scientists," *Prentice Hall*, no. 9th ed., pp. 111-142, 2012.

[2] F. Schaposnik, "Perfect Packing". [Online]. Available: https://www.spoj.com/problems/TAP2015E

[3] G. Micheli, S. Schraven, and V. Weger, "A local to global principle for expected values," Journal of Number Theory, vol. 238, pp. 1–16, 2022.

[4] M. Baron, "Probability and statistics for computer scientists," *Chapman and Hall/CRC*, 2018.

[5] D. A. Bini, G. Latouche, and B. Meini, "Numerical Methods for Structured Markov Chains," *Oxford University Press*, pp. 3-21, 2005.

[6] A. Katz, W. Pakornrat, A. Kau, and E. Ross, "Law of Iterated Expectation". [Online]. Available: https://brilliant.org/wiki/law-of-iterated-expectation

[7] H. Anton and C. Rorres, "Elementary Linear Algebra, Applications Version," *Wiley* , pp. 25-52, 2014.

[8] S. Djanali, "Sistem Digital (Digital System)," *ITS Press Surabaya* , no. 3rd ed., 2015.

[9] N. Kalicharan, "Data Structures in C," *Createspace Independent Publishing Platform* , 2008.

[10] S. Halim and F. Halim, "Breadth First Search (BFS)," *Lulu.com*, no. v. 3, pp. 123-124, 2013.

[11] W. Stallings, "Operating systems: Internals and design principles," *Prentice Hall* , no. 7th ed., pp. 24-31, 2012.

[12] E. Kani, "Geometric Series of Matrices". [Online]. Available: https://mast.queensu.ca/ math211/m211oh/m211oh96.pdf

[13] ——, "Clusters". [Online]. Available: https://www.spoj.com/clusters

[14] ——, "How to cope with SPOJ?". [Online]. Available: https://www.spoj.com/tutorials/USERS.

[15] S. Yendri, R. Soelaiman, U. L. Yuhana, and S. Yendri, "Dynamic Programming Approach for Solving Rectangle Partitioning Problem," *IAENG International Journal of Computer Science*, vol. 49, no. 2, pp. 410–419, 2022.

[16] S. S. Mopuri, "System Design — Online Judge with Data Modelling". [Online]. Available: https://medium.com/@saisandeepmopuri/system-design-online-judge- with-data-modelling-40cb2b53bfeb

[17] S. Yendri, R. Soelaiman, and Y. Purwananto, "Hybrid Algorithm to Find Minimum Expected Escape Time From a Maze," *Engineering Letters*, vol. 31, no.1, pp. 346-357, 2023

[18] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. "Introduction to Algorithms (3rd ed.)," *MIT Press*, 2009.

[19] J. Shi, S. Li, Y. Xu, R. Fu, X. Wang, and T. Wu, "FlashSparse: Minimizing Computation Redundancy for Fast Sparse Matrix Multiplications on Tensor Cores," *arXiv preprint*, arXiv:2412.11007, 2024.