**Efficient use of Randomization Algorithms for Probability Prediction in Baccarat using: Monte Carlo & Las Vegas method**

Avani Jindal[1], Janhvi Joshi[1], Nikhil Sajwan[1], Naman Adlakha[1], Sandeep Pratap Singh[1]

[1] University of Petroleum & Energy Studies (UPES)

Department of Virtualization, School of Computer Science

Dehradun- 248007, Uttarakhand, India

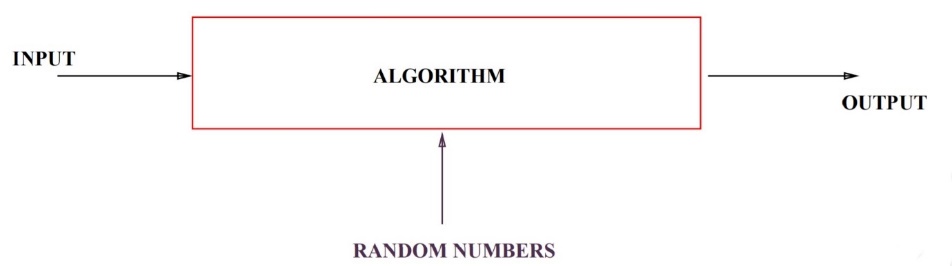
500060696@stu.upes.ac.in

**Abstract:** Randomized algorithms use a degree of randomness in their logic. Such algorithms provide a solution which may not always be optimal, but is often produced faster than a brute force process. This paper explores the two classes of randomized algorithms: Las Vegas and Monte Carlo. A Las Vegas algorithm always produces the correct result but its running time is based on a random value. A Monte Carlo algorithm has a deterministic running time and produces an answer that has a probability of >=1/3 of being correct. By means of this paper, we develop algorithms that predict the winning probabilities of the betting options in the casino table card game of Baccarat. Both classes of algorithms have been implemented in two ways each, varying in space and time complexities. We also propose and implement a novel approach to reduce the time complexity of a typical Las Vegas algorithm through the use of Multithreading. A comparative study for the algorithms has also been done.

**Keywords:** Randomisation algorithms, Las Vegas, Monte Carlo, Multithreading in C, Baccarat, Optimisation

1. **Introduction**

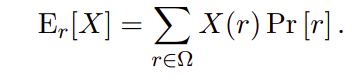
The fields of cryptography, load balancing, and parallel and distributed computing majorly employ randomized algorithms in their implementation. However, these algorithms and their subclasses have not been explored much. A randomized algorithm receives some random values, along with the input data, which are used for making random choices, hence affecting the algorithm’s behaviour. This way, it ultimately achieves a good performance in the average case. Mostly the execution time is a random variable, even when the input given is fixed. We usually talk about the expected worst-case performance of such algorithms—that is, the average time taken when it is given the worst input of a fixed size.



**Fig. 1.** Randomisation Algorithm Illustration

To give a formal definition of a randomized algorithm [1], it can be thought of as a machine M which calculates *M(x,r)*. Here, *x* signifies the problem input and *r* signifies the sequence of random bits. M is a typical random-access machine, which has a memory space and we can perform all operations on a memory location. These arithmetic operations on the integers typically involve read, write or other mathematical operations up to *O(log n)* bits in a given constant time. The assumption that we make here is that constant time is taken in generation of a random integer of size *O(log n).*

The running time of the algorithm and the number of various constant-time operations depends on the *r* random bits. We formally define them as a **random variable**. Let a probability space Ω be defined consisting of all possible sequences *r*, each having a probability *Pr [r]*. On some input *x,* the running time is given as an **expected value** *Er[time(M(x, r))]*. Here, for any X,



For a deterministic algorithm, if we assume it runs in time *O(f(n))*, where *n = |x|*, then we see that the run time is dependent on the size of the input. Whereas, the randomisation algorithm given by us runs in expected time *O(f(n))*. It suggests that *Er[time(M(x, r))] = O(f(|x|))* for all inputs *x*.

The fact stated above is dissimilar from the conventional worst-case analysis, where no *r* and no expectation is defined, and average-case analysis, where again no *r* is defined and the expectation is only over some distribution on *x*.

1. **Methods to generate randomness**

There are typically two methods for generating the random bits r, which are essential inputs for any randomized algorithm. True random numbers are values generated through using some type of physical phenomenon as mentioned in [2],[3] and [4]. Another method includes the generation of pseudo random numbers based on a **seed** value. The seed itself is chosen at random, often times using physical randomness. Other parameters including quantum random walks [5] are also used. Pseudo random numbers can be cryptographically-secure, but for the purpose of this paper, we have chosen the statistical pseudo-randomness method to generate the needed random inputs since security of the application was not the main focus of the research.

1. **Classification of Randomisation Algorithms**

**Monte Carlo** Monte Carlo is that class of algorithm which may return the correct result or the incorrect result with some probability. The algorithm resources used in this are often bounded and thus it has a deterministic or fixed run time. It gives better probability results when it is run for a larger number of iterations. This class can also be used to predict the value of Pi. Las Vegas algorithms, which we will discuss further, are said to be a subset of Monte Carlo.

**Las Vegas** A Las Vegas algorithm ​ returns the correct or optimum result always, and informs when it fails. Its run time differs at each run since it depends on a random value, even for the same input. When László Babai introduced this term, he explained that the algorithm could be thought to depend on a series of coin flips to determine its next step. A classic example of this class of algorithms is the Randomized Quicksort wherein the pivot​ p in the algorithm is chosen randomly from the array of elements.

1. **Pthread Library**

A standardized programming interface to provide the full capabilities of threads in UNIX systems was provided by the IEEE POSIX 1003.1c standard (1995). POSIX threads or Pthreads are the thread implementations which adhere to this standard. Implementation of pthread is available with GCC compiler and has been used for multithreading. We have implemented all the algorithms in C language with a GCC compiler only.

1. **Baccarat**

Baccarat is a game of cards which contains 8 decks. There are 416\*415\*414\*413 possible cases in first hand which makes it almost impossible to be followed for patterns/predictions. To be able to even calculate the probabilities with a great deal of accuracy, it takes a large amount of time. Hence, it served as a suitable problem statement that could be solved by our algorithms. The game of Baccarat is dealt from a shoe containing 1 to 8 decks of 52 cards each. Tens and face cards are counted as zero. A hand of two cards is dealt for both the banker and the player alternately. Bets are placed on the banker's hand, or on the player’s hand, or on the tie. The hand whose sum of the face values of the cards is closest to 9 wins. If the hands have the same value, the result is a tie.

The rest of the paper is organized as follows. First, a brief outline about the work that has been done in the recent years related to these algorithms is given (section 2). In section 3, the proposed approach for the design, implementation and optimization of Las Vegas and Monte Carlo algorithms has been described. An explanation of the game of Baccarat has also been given. This is followed by the analysis of the results obtained in section 4. Finally, in section 5, a comparison of all the proposed algorithms with each other has been done.

1. **Related Works**

When there is an attacker who intentionally tries to give bad inputs to the algorithms, such as Prisoner’s dilemma, randomization algorithms prove to be very useful. It also has various applications in cryptography. The numbers that have to be chosen in cryptographic applications cannot be pseudo-random as they can then be predicted by the attacker. Therefore, the source of numbers should be truly random. Another important area which uses the concept of randomness is quantum computing.

A randomization algorithm was first used by Michael O. Rabin as a method for closest pair problem in computational complexity. After the discovery of a randomized primality test, the study of these algorithms took a great leap.

There are different fields in which Monte Carlo and Las Vegas algorithms have been put to use. In [6], randomization algorithms have been used for computations of high dimensional Gaussian weighted integrals. The integrals have been calculated using Markov chain Monte Carlo and then they are compared. This approach can also be generalised and then used to compute integrals by general weighted functions other than the Gaussian weights. Other cases where Monte Carlo has been used is to approximate the energy cost of a problem instance [7]. It is a very complex scenario to do this because of the impact of distribution of runtimes. This approach considers Weibull and Pareto distributions, the two common runtime continuous runtime distributions. It demonstrates the interesting and uncommon relationship between parallelism, runtime and energy cost in combinatorial solving. Another research related to this is [8], where the tolerance analysis is formulated mathematically. It simulates the effects of geometrical derivations on the geometrical behaviour of the mechanism. To perform this computation, two approaches based on Quantified Constraint Satisfaction Problem solvers and Monte Carlo simulation have been suggested and tested.

One of the closely related works is the probabilistic model for parallel execution of Las Vegas algorithms [9]. It analyses the runtime distribution of sequential runs of the algorithm. While their approach is to speed up the algorithm by analysis of gathered data, we use the concept of multithreading for this. Another work on Las Vegas is for evaluating the performance of the algorithm based on identification of empirical run-time distributions [10]. They demonstrate their approach by applying it to Stochastic Local Search (SLS) algorithms for the satisfiability problem (SAT) in propositional logic. They have even discussed the pitfalls caused due to use of improper methods and the benefits of the given approach. A lucid and universal strategy with a property is presented in [11]. For any algorithm A,



where *scL* is a strategy and *T*(*A*, *scL*) is the expected value of the running time of the simulation of *A* under strategy *scL*. The application of this method can also be found in Markov processes where we get the correct results after termination of an iterative method [12], in identifying the error in circuits of logic gates using partitioning algorithms [13] and also in communication networks for optimal routing in a topology to fulfil the time constraints with the use of clustering algorithms [14].

1. **Proposed Approach**
   1. **Using Monte Carlo method**

A simple example to explain this class is given below:

*Algorithm:*

*repeat 300 times:*

*k= RandInt(n)*

*if A[k]=1*

*return k*

*return “Failed”*

The above example will run 300 times, which is the number of iterations specified, hence leaving a possibility that, within all the 300 times, the algorithm might not find “1” in the array. Therefore, Monte Carlo wagers with the correctness of the result, rather than the run-time.

In our application, we checked for a given number of chances in baccarat with the given shoe(8 decks) and for each chance we calculated a random outcome using a baccarat simulation,

*Algorithm:*

*for i in (0,n)*

*check (randcard());*

*Increase counters(player,banker,tie,chances) appropriately*

*Calculate and show probability to user of all outcomes;*

Since we cannot count the same card combination more than once, we create a flag array of size equal to the total number of cards in the shoe (416 in our case). Each time a card A is randomly chosen, we change the value of the array[A] from 0 to 1, indicating that this card must not be chosen again. Since we are simply accessing the values through its index, the operation doesn’t take too long.

*Mathematical Representation:*

*Let N be number of test cases taken before result is shown,*

*For each iteration 1 to N,*

*let C1, C2, C3, C4 be randomly generated cards from available shoe =>*

*shoe = Values [0-415] – Used Set {} (null at start of game)*

*let a, b, c, d be face values of C1, C2, C3, C4 respectively,*

*our operations can be seen in the equation given below*

*Total Probability = 1 = (N∑ Player Wins (a, b, c, d))/N +*

*(N∑ Player Wins (a, b, c, d))/N + (N∑ Player Wins (a, b, c, d))/N*

Note: each summation separated by ‘+’ operator is a probability to be shown in output.

* 1. **Using Monte Carlo method with Data Structures**

In the second version of Monte Carlo with data structures we used arrays of linked lists connected hierarchically to form a 416-size tree like structure. Each node is kept null until called and each leaf node represents a unique case of hand, if while traversing the nodes for cards a leaf node comes across as ‘not null’, it is a sign that that particular combination of cards has already been discovered in previous iterations and instead the algorithm starts looking for a new combination, hence increasing the accuracy in the resultant probability. This is done because all combinations taken in account for the calculation must be unique.

*Algorithm:*

*for i in (0,n)*

*Get 4 randcard();*

*traverse node forest for 4 cards*

*if 4 cards exist in forest*

*i--;*

*continue;*

*else*

*initialise nodes;*

*check(4 cards);*

*increase counters(player,banker,tie,chances) appropriately;*

*calculate and show probability to user of all outcomes;*

This algorithm takes more time and memory space than the typical implementation of Monte Carlo described above because creating and then searching in a tree is expensive.

*Mathematical Representation:*

*Let N be number of test cases taken before result is shown,*

*For each iteration 1 to N,*

*let C1, C2, C3, C4 be randomly generated cards from available shoe which are unique together=>*

*shoe = Values [0-415] – Used Set {} (null at start of game)*

*let a, b, c, d be face values of C1, C2, C3, C4 respectively,*

*our operations can be seen in the equation given below*

*Total Probability = 1 = (N∑ Player Wins (a, b, c, d))/N +*

*(N∑ Player Wins (a, b, c, d))/N + (N∑ Player Wins (a, b, c, d))/N*

Note: each summation separated by ‘+’ operator is a probability to be shown in output.

* 1. **Using Las Vegas method**

To demonstrate this class of algorithm, we use the following example:

*Algorithm:*

*repeat:*

*k=RandInt(n)*

*if A[k]==1*

*return k;*

An array A is indexed by a variable ​ k, which is randomly generated. The value ​k is returned if the index contains the value ​ 1. Otherwise, this process is repeated until 1 is found​. This Las Vegas Algorithm finds the correct answer on all occasions, but because of the randomization, it does not have a fixed runtime; there is a possibility that a huge amount of arbitrary time elapses before the termination of the algorithm.

As we know, Las Vegas runs for an indefinite time span but always gives the correct result. In our approach, we implemented this algorithm in two ways: using the brute force technique and by using the concept of multithreading in C language only. We shifted from brute force technique to multithreading because of extremely high time complexity in the former.

In the brute force technique, we iterated through all the remaining card combinations in the deck sequentially and compute the win or loss in each possible case. After that, we calculated the probability of the cases in which the banker won, in which the player won and in which there was a tie. The calculated probabilities were then given to the user and he could play his next move accordingly. The algorithm we used is as follows-

*Algorithm:*

*for i in (0,416):*

*if card\_is\_drawn, continue;*

*for j in (0,416):*

*if card\_is\_drawn, continue;*

*for k in (0,416):*

*if card\_is\_drawn, continue;*

*for l in (0,416):*

*if card\_is\_drawn, continue;*

*checkWin(i,j,k,l)*

*Mathematical Representation:*

*shoe = Values [0-415] – Used Set {} (null at start of game)*

*total = total combination possible from the available shoe*

*let C1, C2, C3, C4 be randomly generated cards from available shoe which are unique together=>*

*let a, b, c, d be face values of C1, C2, C3, C4 respectively,*

*our operations can be seen in the equation given below*

*for 1 to total chances =>*

*Total Probability = 1 = (total∑ Player Wins (a, b, c, d))/total +*

*(total∑ Player Wins (a, b, c, d))/total + (total∑ Player Wins (a, b, c, d))/total*

Note: each summation separated by ‘+’ operator is a probability to be shown in output.

* 1. **Using Las Vegas method with Multithreading**

As we can see, normal Las Vegas is done using four for loops which is not an optimal approach. This also increased the execution time. So, as a solution to this problem, we chose to switch to multithreading. As discussed in the Introduction section, a thread is a lightweight process. Using them, we can make use of the modern-day microprocessor architecture and also make the program faster. It is possible to do in C by using the <pthread.h> library.

*Algorithm:*

*create 416 POSIX threads;*

*for each\_thread:*

*for j in (0,416):*

*if card\_is\_drawn, continue;*

*for k in (0,416):*

*if card\_is\_drawn, continue;*

*for l in (0,416):*

*if card\_is\_drawn, continue;*

*checkWin(tid,j,k,l)*

So, it can be seen that we created 416 threads that take up the workload of one of the four for loops completely reducing the time complexity by a whole power extent. Each thread works concurrently and this fastens the processing. All the possible card combinations are tested using the three for loops and the unique thread id assigned to each thread.

*Mathematical Representation:*

*shoe = Values [0-415] – Used Set {} (null at start of game)*

*total = total combination possible from the available shoe*

*threadload = total/threads*

*let C1, C2, C3, be randomly generated cards from available shoe which are unique together=>*

*let a, b, c be face values of C1, C2, C3, respectively,*

*our operations can be seen in the equation given below*

*for 1 to threadload chances =>*

*threadX = (threadload∑ Player Wins (a, b, c, d))/**threadload + (threadload∑ Player Wins (a, b, c, d))/total + (threadload ∑ Player Wins (a, b, c, d))/ threadload*

*Here X is sequentially taken from shoe such that*

*shoesize ∑ threadx = 1*

Note: each summation separated by ‘+’ operator is a probability to be shown in output.

* 1. **Simulation of Baccarat**

Our proposed algorithms find the probability of winning in the card game of baccarat. The flowchart explains the flow of how the algorithm simulating the game works.

Choose algorithm to be used

Place bet on banker, player or tie

Display cards of both hands and winner of that round

Display probability of winning of each hand for next round

Repeat till end of game

**Fig. 2.** Flowchart of Baccarat Simulation

The random() function in C has been used to generate random cards values like the dealer would choose from a shoe of cards. The face value of the cards is calculated and then the sum of the cards of banker’s and player’s hand is taken. One thing to note here is that in baccarat, if the face value of a card is more than 9, then it is considered to be 0.

A few rules need to be followed in baccarat regarding the calculation of sum of the cards: if the sum of player hand is greater than 5 and the banker hand is lesser than 6, then the banker needs to draw one more card from the deck. If the vice versa happens, then the player needs to draw an additional card. The face value of this additional card is also added in the final sum now. The hand with the highest face value sum wins the current hand. If the sum is equal, then it is a tie. In this paper, we have considered a simulation with a shoe containing 8 decks of cards.

1. **Result Analysis**

To analyse the results and performance of our algorithms, we run the simulation of Baccarat game. The total number of combinations that the application checks is around 12,000 crores. First of all, it displays a list of the four algorithms that can be used to calculate the probability of winning in the next round. We choose one of them to do so and get the required result. On selecting Monte Carlo, the calculation takes approximately 1 second. It considers 20 crore unique combinations randomly. Monte Carlo using data structures runs for approximately 4 seconds. It takes more time than the normal one because of the time overhead associated with allocating linked lists and traversing them in each iteration. The brute force Las Vegas algorithm takes a long time of around 6 hours to compute all the 12,000 crore combinations sequentially. Optimised Las Vegas which uses multithreading shows an exponential decrease in the execution time of approximately **45 seconds**. Parallel threads help in this case.

The probabilities output from each algorithm and shown in the table below-

**Table 1.** Resultant Probability Predictions

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Probability of (~) | | |
| Banker | Player | Tie |
| Monte Carlo | 43.87% | 43.85% | 12.23% |
| Monte Carlo using Data Structures | 44% | 43.8% | 12.16% |
| Las Vegas | 46.5% | 45.9% | 7.6% |
| Las Vegas using multithreading | 46.15% | 46.01% | 7.83% |

To verify our results and ensure the correctness, we looked up the standard range of these probabilities [15] and found them to be as follows:

Probability of Banker~ 45.85%

Probability of Player~ 44.62%

Probability of Tie~ 9.53%

Therefore, we can see that the results given by our algorithms are in close accordance with the standard approximate probabilities. The user can further place his bet on either the banker, player or tie and verify the probability of winning in each subsequent round.

1. **Comparison of Algorithms**

Four algorithms were designed to compare Monte Carlo and Las Vegas methodologies, two were vanilla versions Monte Carlo Algorithmic Unit (MCAU) and Las Vegas Algorithmic Unit (LVAU). Following their normal protocols, MCAU, as expected gave us high performance, with the algorithm giving results within a second. Its time complexity is O(n) where n is number of cases chosen randomly. The faster speed costs us with some errors (2%-5%). LVAU, on the other hand, gave dismal performance going through all cases (≈ 30B) needing hours to resolve each hand, with a time complexity of O(n^4) where n is the number of cards left in the shoe. However, at the cost of performance time, we get results with no inaccuracy.

It was clear from vanilla testing that MCAU works better as the probability distribution in card games seems to be normalised throughout the deck making it ideal for applications requiring fast processing, for example Big Data Analytics. However, as much as the error percentage may seem small for one hand, on consecutive applications the error percentage can compile to give erroneous results in mission critical applications where even such minute errors are unacceptable, for example, Pharmaceuticals, Molecular Training and Deep learning.

We designed two more algorithms each overcoming the drawbacks of the Vanilla versions above, while still adhering to the protocols of both the algorithms - Monte Carlo with Data Structure (MCAUDS) and Las Vegas with Threading (LVAUTH).

MCAUDS uses an array of linked lists to form a forest data structure to memorise card combination choices considered before by the algorithm and not repeating them hence increasing accuracy of the random choices. It was observed that in this case, MCAUDS took time to allocate and create the array of linked lists. Creating the forest for a number of test cases as small as 20,000 consumed large amounts of memory and time. Since LVAUTH leverages multithreading, the performance in terms of time, for the same amount of test cases as in LVAU, increases manifold. The time taken by this algorithm has reduced from hours to seconds, although increasing the memory consumption.

**Table 2.** Comparison of the algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best Case Time Complexity | Worst Case Time Complexity | Space Complexity |
| Monte Carlo Algorithmic Unit | Ω(n) | O(n) | 877bytes |
| Monte Carlo Algorithmic Unit using Data Structures | Ω(n) | O(n^2) | 2626bytes |
| Las Vegas Algorithmic Unit | Ω(n^4) | O(n^5) | 920bytes |
| Las Vegas Algorithmic Unit using Threads | Ω(n^3) | O(n^4) | 26658bytes |

1. **Conclusion and Future Scope**

There are many methods to calculate probability in various fields. These classes of algorithms have not been explored much till date as they combine the concepts of randomisation and probability determination. As mentioned in the related works, they have been used in some other fields but not in this one. We could predict the probabilities in less than a minute for such an enormous number of possible cases using them. The user can have the probabilities before each round and place his bet accordingly. This will increase his chances of winning in each round. The prediction of the algorithm also improves after each round as more and more cards are withdrawn from the deck. Implementation of multithreading helped in achieving an execution time which was approximately 99% lower than the brute force technique. This enhancement in performance is one of the most remarkable milestones in our research. Having done this, we can utilise the same in varied fields of research which involve a larger number of computations. This will give us optimal results in a few seconds.

Other than this, a similar approach can be used in other card games like Black Jack, Poker and so on. We can also use these for various other applications like weather forecasting, sports games predictor, insurance options prediction, etc.

1. **References**

[1] James Aspnes: Notes on Randomized Algorithms CPSC 469/569: Fall 2019, Creative Commons Attribution-ShareAlike 4.0 International license.

[2] Cherkaoui A., Fischer V., Fesquet L., Aubert A. (2013) “A Very High Speed True Random Number Generator with Entropy Assessment”. In: Bertoni G., Coron JS. (eds) Cryptographic Hardware and Embedded Systems - CHES 2013. CHES 2013. Lecture Notes in Computer Science, vol 8086. Springer, Berlin, Heidelberg

[3] Li Dejun, Pei Zhen, “Research of True Random Number Generator Based on PLL at FPGA,” Procedia Engineering, Volume 29, 2012, Pages 2432-2437, ISSN 1877-7058.

[4] Barak B., Shaltiel R., Tromer E. (2003) “True Random Number Generators Secure in a Changing Environment”. In: Walter C.D., Koç Ç.K., Paar C. (eds) Cryptographic Hardware and Embedded Systems - CHES 2003. CHES 2003. Lecture Notes in Computer Science, vol 2779. Springer, Berlin, Heidelberg

[5] Yang, Y., Zhao, Q. “Novel pseudo-random number generator based on quantum random walks.” *Sci Rep* 6, 20362 (2016).

[6] Z. Zhao and M. Kumar, "A comparative study of randomized algorithms for multidimensional integration," 2012 15th International Conference on Information Fusion, Singapore, 2012, pp. 2236-2242.

[7] Siala M., O'Sullivan B. Combinatorial search from an energy perspective, Information Processing Letters, Volume 148, 2019.

[8] Jean-Yves Dantan, Ahmed-Jawad Qureshi, “Worst-case and statistical tolerance analysis based on quantified constraint satisfaction problems and Monte Carlo simulation,” Computer-Aided Design, Volume 41, Issue 1, 2009, Pages 1-12, ISSN 0010-4485.

[9] C. Truchet, F. Richoux and P. Codognet, "Prediction of Parallel Speed-Ups for Las Vegas Algorithms," 2013 42nd International Conference on Parallel Processing, Lyon, 2013, pp. 160-169.

[10] Hoos, Holger H.  Stutzle, Thomas, “Evaluating Las Vegas Algorithms - Pitfalls and Remedies,” arXiv e-prints, 2013, January 01, 2013, Computer Science - Artificial Intelligence.

[11] Michael Luby, Alistair Sinclair, David Zuckerman, “Optimal speedup of Las Vegas algorithms,” Information Processing Letters, Volume 47, Issue 4, 1993, Pages 173-180, ISSN 0020-0190.

[12] R. Fujita, K. Iwata and H. Yamamoto, "An Iterative Algorithm to Optimize the Average Performance of Markov Chains with Finite States," 2019 IEEE International Symposium on Information Theory (ISIT), Paris, France, 2019, pp. 1902-1906.

[13] I. Scarabottolo, G. Ansaloni, G. A. Constantinides and L. Pozzi, "Partition and Propagate: an Error Derivation Algorithm for the Design of Approximate Circuits," 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2019, pp. 1-6.

[14] C. Hsu, S. Hung, H. Chen, F. Sun and Y. Chang, "A DAG-Based Algorithm for Obstacle-Aware Topology-Matching On-Track Bus Routing \*," 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2019, pp. 1-6.

[15] Best-Baccarat-Online.com Homepage, http://www.best-baccarat-online.net/baccarat-odds-player.html, last accessed 19/02/2020