Analysis of N-Queens Stochastic Algorithms Using Probabilistic Model Checking

Ronald Picard, Bernard Serbinowski
Vanderbilt University
Nashville, TN, USA

***Abstract*—Probabilistic model checking extends model checking to domains in which actions will be taken based on a certain probability. This aligns nicely with modeling many real world systems in which actions are taken in a stochastic manner. Unfortunately, probabilistic model checking further complicates the already high computational complexity challenges associated deterministic model checking. N-Queens is a commonly studied computer science and mathematics problem. There are many algorithms that solve this problem under different constraints. Some algorithms are iterative and deterministic, while other algorithms are stochastic in nature. In this paper we present a probabilistic model checking approach to check a fundamental property of these algorithms; namely, there ability find a solution within a given number of transitions (iterations). This number is not fixed, but rather a probability of successful completion within some number of transitions. We utilize PRISM, a probabilistic model checker, to model these algorithms and find the probability that this property is satisfied after I interactions. We present and discuss some relevant challenges associated probabilistic model checking and make some suggestions for future research.**

***Index Terms*—stochastic analysis, probabilistic model checking, verification, N-Queens**

I. Introduction

Probabilistic model checking extends deterministic model checking to modern domains in which a system's actions are taken only taken with a certain probability. Many real world systems possess transition systems that are stochastic rather than deterministic in nature. Unfortunately, when moving from deterministic model checking to probabilistic model checking there are further complications arising from increased computational complexity. In order to examine these challenges, along with the utility of probabilistic model checking, we consider the N-Queens problem.

N-Queens in a generalization of a common mathematics and computer science problem called Eight Queens. The problem involves the positioning of N queens on an N by N chess board. The goal of the problem is to transition the queens into new positions until no queen can attack another. Many stochastic algorithms exist to solve this problem. Each algorithm can be broken down into a stochastic transition system. A potentially interesting point of analysis for these algorithms is determining the probability with which they will find a solution within I transitions (iterations). This analysis can be performed by developing a formal model of a stochastic transition system and analyzing it in a probabilistic model checker. PRISM is a probabilistic model checker that is capable of performaing such an analysis.

In this work we present a probabilistic model checking analysis of several stochastic algorithms capable of solving N-Queens using PRISM. Similar work has been done in a PRISM case study [1] based on the well known Dining Philosophers problem. The problem involves N philosophers sitting around a round table. There is a bowl of rice for each philosopher, and N chopsticks shared among all the philosophers. A philosopher requires both their right and left chopstick to eat. A hungry philosopher may only eat if both their chopsticks are available, otherwise the philosopher must put their chock sticks down and begin meditating again. The goal of an algorithm is to allow each hungry philosopher a chance to each. In the case study a stochastic algorithm created by [2] is modeled and analyzed. A resulting plot from the case study is shown in Figure 1. The results provide the probability that the algorithm will arrive a solution as a function of the number of transitions, K, for varying numbers of philosophers, N.

We present a similar approach to modeling and analyzing N Queens using PRISM. We discuss the challenges and complexities associated probabilistic model checking, as well as provide recommendations for future research.

II. Modeling Framework

In this section we introduce some fundamental topics required to understand the different sections of this work.

1) N-Queens: N-Queens is a generalization of a common computer science optimization problem called Eight Queens. This problem involves N number of Queens on and N by N chess board. The goal of the to find a solution configuration in which no queen can attack (see Figure 4) another from their current position. See Figure 3 for an illustration of what a solution looks like for the variant of 8-Queens. There are many deterministic and stochastic algorithms capable of solving N-Queens under specific constraints. We focus on a subset of stochastic algorithms. We model and analyze these algorithms in a probabilistic model checker for the probability that a solution is found based on the number of transitions taken. To constrain the problem space and achieve comparable results, we initialize the queens along the diagonal configuration as illustrated by Figure 2. In addition, we only consider algorithms that move a given queen along a row to a new column space, and exclude algorithms that move a queen along a column to a new row space.

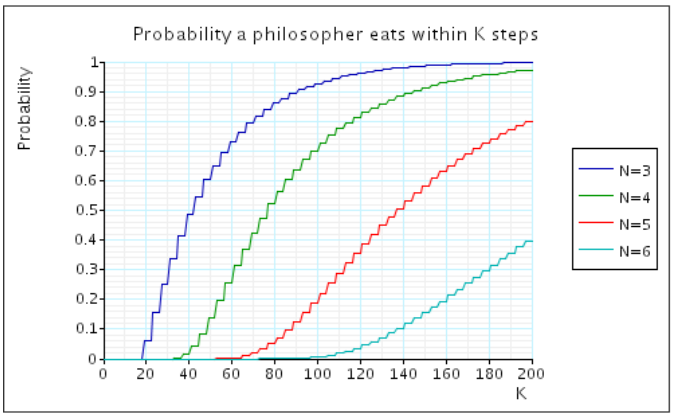

Fig. 1: PRISM Results for a Dining Philosophers Algorithm [1]

Fig. 2: Initial Configuration of 8 X 8 Board With 8 Queens [3]

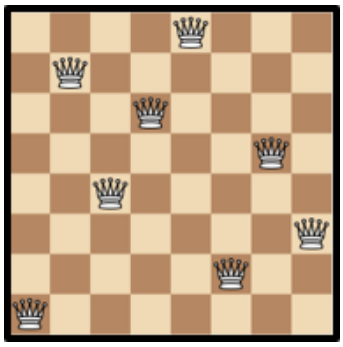

Fig. 3: One Solution Configuration of 8 X 8 Board [3]

Fig. 4: Illustration of a Queen's Attack Paths [3]

A. Algorithm: Random Column Move

Strictly speaking, this does not operate on a random column. Rather, the algorithm selects a queen which is presently under attack and moves the queen to a different column chosen at random. [4] As such, the algorithm naturally terminates when a solution is found, as no queens are under attack when a solution is reached. Since this algorithm moves an attacked queen to a random new column without considering whether or not there is already a queen in that column, checking if a solution is reached after a given transition requires checking both columns and diagonals for possible attacks. Note that checking rows is unnecessary, because all the queens start in distinct rows and each queens row is never changed.

B. Algorithm: Random Column Swap

In a similar manner to the II-A algorithm, this algorithm also only operates on an attacked queen. Unlike the II-A algorithm, this one operates on two queens at once. Let $Q1$ denote the queen under attack and $Q2$ a different queen, selected at random. Next, the two queens are swapped, by which we mean that $Q1$ is moved to the column which $Q2$ occupies and $Q2$ moves to the column that $Q1$ occupied. Note that the rows the queens occupy remain unchanged. Furthermore, $Q2$, unlike $Q1$, may or may not be under attack. We do not check. [4] Please note that since all queens start in different columns and rows (see Figure 2), and the only operation which is performed upon the queens is a column swap, at every point during the algorithm all queens are in different rows and columns. As such, checking if queens are attacking each other involves only checking for diagonal attacks in this case.

C. Algorithm: Simulated Anneal Swap

This algorithm is somewhat more complex. In general, it operates in the same way that the Random Column Swap does. However, instead of accepting any swap it attempts to only allow good swaps. Thus, if a swap would result in less queens being attacked, it is always taken; however, if a swap results in more queens being attacked, we take it with probability p. This probability decreases as the number of successful swap transitions taken (iterations) increases. [4] In our case, it starts at 1, and decreases by $1/100$ after each successful swap to a min of $1/100$.

1) Probabilistic Model Checking: Probabilistic model checking is a form of model checking involving a stochastic transition system. A stochastic transitions system in extension of deterministic transitions system in which, if guard condition is satisfied, then one of several transitions is taken based their respective probabilities. We use probabilistic model checking to find the probability that a property is satisfied after N number of stochastic transitions. In order to do this, probabilistic model checkers provide a formal language in which to formally specify a stochastic transition system. These model checkers then build a transition graph utilizing discrete-time Markov chains (see subsection II-C2) and calculate the probability that a property is satisfied after a given number of iterations.

2) Discrete-Time Markov Chains (DTMCs): DTMCs are chains of probabilistic transitions that occur under the Markov property. The Markov property is that, the probability that a transition is taken, is only a product of the current state. [5] See Figure 5 for an example DTMC. At a state, for example S_0, there is a set of probabilities that each correspond to the likelihood that a transition is taken. Then in the next state, for example S_1, there is a new set of probabilities that each correspond to the likelihood that a transition is taken from that state. DTMCs can be used to describe chains of probabilistic transitions for a stochastic transition system. Using this approach, we can run simulations to find the likelihood that we have reach a solution state in our algorithms given I number of transitions (iterations).

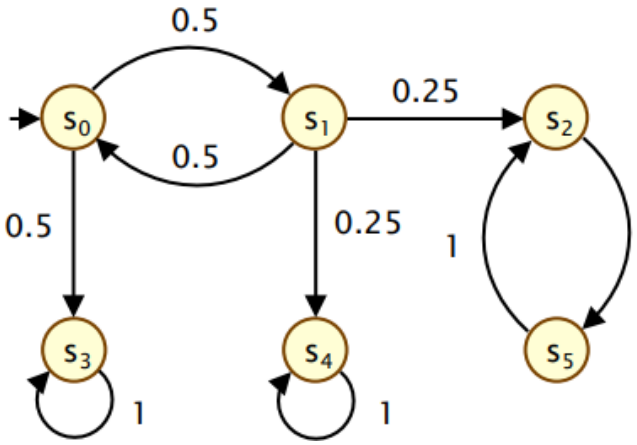

Fig. 5: Example Discrete-Time Markov Chain (DTMC) [6]

D. Computational Tree Logic (CTL)

Computational Tree Logic (CTL) is a subset of first-order logic related to temporal operators that allow for the evaluation of predicates over traces (or paths), and trace quantifiers. Each element of a trace corresponds to a state at a specific time. Common CTL operators are listed below. [7]

1) Temporal Operators:
   a) F ϕ: ϕ holds eventually (for a future element of a trace)
   b) G ϕ: ϕ holds globally (for for all elements of a trace)
   c) X ϕ: ϕ holds next (for the next element of a trace)
   d) ϕ_1 U ϕ_2: ϕ_1 holds at each element of the trace until ϕ_2 holds
2) Trace (or Path) Quantifiers:
   a) A F ϕ: ϕ holds in the future for all traces ψ
   b) E F ϕ: ϕ holds in the future for some traces $\lambda \subseteq \psi$

E. Probabilistic Computational Tree Logic (PCTL)

Probabilistic computational tree logic (PCTL) is an extension of computational tree logic (CTL), providing the addition of a probabilistic trace operator, P. This additional operator provides the probably that a temporal operator formula will be satisfied. This operator provides the necessary and sufficient description needed to specify the probability that a solution state is reached within a trace. Some common uses of the P operator is listed below. [7]

1) $P > P_{threshold}$ [F ψ]: The probability that ϕ holds in the future traces is greater than $P_{threshold}$.
2) $P <= P_{threshold}$ [F ψ]: The probability that ϕ holds in the future traces is less than or equal to $P_{threshold}$
3) $P = P_{threshold}$ [F ψ]: The probability that ϕ holds in the future traces is equal to than $P_{threshold}$

1) PRISM: PRISM is a probabilistic model checker that supports the use of DTMCs and PCTL in order perform various types of probabilistic model checking. PRISM supports simulated runs through DTMCs which can verify PCTL formulas, and experiments which can help find, $P_{threashold}$, of PCTL formals; such as the probability that a solution is found in the future. [8]

2) PRISM Experiments: PRISM supports experiments which can find the probabilities for which the PCTL formulas are satisfied over ranges of values. Using a counter as the range of values, we can use PRISM to find the probability that a system property is satisfied in a given number of transitions (iterations). [8]

III. Formal Modeling

A. Meta-Scripts for PRISM Models Structure

We develop Python meta-scripts for generating PRISM's formal model files for each algorithm up to N number of queens. This provides scalability that the PRISM models do not have innately.

B. Formal Models

The model files for PRISM are written in primarily declarative specifications in which the stochastic transition system is declared, along with accompanying variables and formulas, then evaluated using the internal procedures of PRISM simulations and experiments. See Figure 12 for an example of the PRISM model window with the 4 queens random column swap model loaded. As seen in Figure 12, there are a handful of variables, follow by a module block that encompasses the stochastic transition system specification. Following the module block are two formula specifications need for evaluation of the guard conditions and our property specification (see subsection III-G). Last, we see three module renaming lines. Essentially, these lines duplicate the transition system for queen 1 to the relative transition system for queens 2, 3, and 4 so that they do not have to be specified directly.

C. Specifications

There are a few common specifications for each formal PRISM model. First the queens positions in the PRISM models are modeled as global variables with integers ranging from 1 to N for y (column) positions . The queens are initialized along the diagonal of the chess board as shown in Figure 2. Note that a single integer is sufficient to encode the position of each queen, as their row numbers are static. We do, however, include an appropriately named row constant for each queen, to improve readability. Each PRISM model is accompanied

with a module that specifies the stochastic transitions for queen 1, which is the queen occupying the top row. This module is then pseudo-copied and renamed (there is a simple technique within the PRISM language for this named module renaming) so that each queen shares a similar stochastic transition set. Finally, each model has a set of formulas that provide reusable results for the stochastic transitions. See III-B for more information on the formal models. In each algorithm, a transition is only taken if the current queen is under attack; therefore, every model shares the same formula (structurally) for deciding whether a queen is attacked as illustrated by equation 1. Because our initial configuration is along a diagonal, we do not initially have any queens sharing a row or column. In addition, we mostly consider algorithms in which the transitions swap the queens; therefore, we only need to check if the queens are attacked along a diagonal. This is accomplishing by checking if there is an equivalent distance between X and Y coordinates of any two queens. The random column move algorithm is the only non-swap based algorithm we used in this work. Because this algorithm is non-swap based, queens can occupy the same columns and an additional specification is required to determine a queen is under attack on a column, as illustrated by equation 2. Note we have slightly modified the included equations for readability within this report.

$$\begin{gathered} Is\_Queen1\_Attacked?= \\ (((q1x - q2x)=(q1y - q2y)\vee \\ (q1x - q2x)=-(q1y - q2y))\vee \\ ((q1x - q3x)=(q1y - q3y)\vee \\ (q1x - q3x)=-(q1y - q3y))\vee \\ \cdots\vee \\ ((q1x - qNx)=(q1y - qNy)\vee \\ (q1x - qNx)=-(q1y - qNy))) \end{gathered} \tag{1}$$

$$\begin{gathered} Is\_Queen1\_Attacked?= \\ (((q1x - q2x)=(q1y - q2y)\vee \\ (q1x - q2x)=-(q1y - q2y))\vee \\ ((q1x - q3x)=(q1y - q3y)\vee \\ (q1x - q3x)=-(q1y - q3y))\vee \\ \cdots\vee \\ ((q1x - qNx)=(q1y - qNy)\vee \\ (q1x - qNx)=-(q1y - qNy))\vee \\ (q1x{=}q2x)\vee(q1x{=}q3x)\vee \\ \cdots\vee \\ (q1x{=}qNx)) \end{gathered} \tag{2}$$

D. Random Column Move Models

Because this model does not utilize swaps, we must also consider the case when a queen is under attack by another queen in the same column. Fortunately, 2 is sufficient for this purpose. Armed with this slightly longer formula, we are able to determine if a queen in this model is under attack, and therefore able to take the appropriate move action.

Equation 3 shows one example stochastic transition in the case of four queens. This transition states that if the queen is under attack from a diagonal or column and the queen is in column 1, than we move this queen to another column. Each column has a $\frac{1}{3}$ chance of being selected. Note that since this transition is column specific, queen 1 has a total of four such transitions in this case, one for each column. Furthermore, note that every single queen has their own version of these transitions. When more than one guard condition is satisfied, either transition may be taken.

It should be noted that 'under_attack_diag' is the name of a formula. Formulas in PRISM function like include statements in C; that is to say, PRISM copies and replaces the name of the formula with it's value.

$$\begin{gathered} (under\_attack\_diag|under\_attack\_column)\&(q1x = 1) -> \\ \tfrac{1}{3} : (q1x' = 2) + \tfrac{1}{3} : (q1x' = 3) + \tfrac{1}{3} : (q1x' = 4); \end{gathered} \tag{3}$$

E. Random Column Swap Models

The transition model here is quite simple. If a queen is under attack, swap it with another queen at random. Repeat until no transitions are enabled, meaning no queens are under attack.

Equation 4 show one stochastic transition in the case of four queens. This transition states that if queen 1 is under attack, we should swap it with another queen. Each other queen has a $\frac{1}{3}$ chance of being selected for the swap. This is the only transition in queen 1's module, but each queen has it's own module, so there are three other transitions like this one.

$$\begin{gathered} under\_attack -> \tfrac{1}{3} : (q1x' = q2x)\&(q2x' = q1x) \\ +\tfrac{1}{3} : (q1x' = q3x)\&(q3x' = q1x) \\ +\tfrac{1}{3} : (q1x' = q4x)\&(q4x' = q1x); \end{gathered} \tag{4}$$

F. Simulated Annealing Swap Models

This model features several changes. First, we introduce some state variables which are not related to the position of queens on the board, but to the transitions we are taking. This allows us to perform actual model transitions over several steps, which makes the update logic substantially simpler, though it does spread it across several transitions. The steps include:

1) Swap selection. We begin by picking a queen which is under attack, which will be called $Q1$. A random queen $Q2$ is also selected. Note that they cannot be the same queen.
2) Current Attack Values stored. We then store the current total number of queens attacked by $Q1$ and by $Q2$. Note that a single queen may be under attack from multiple sources.
3) Execution of the swap. $Q1$ and $Q2$ swap column positions.
4) Swap Evaluation. At this point, we compare the current Attack Value of $Q1$ and $Q2$ to the previous Attack Values we have stored. If the sum is smaller, this is our new state. If it is larger, we might have to revert.
5) Possible revert. Just swaps the queens back to their original positions.Furthermore, rather than simply checking if a queen is under attack, we compute how many queens are attacking this queen. This does not change the nature

of the formula, though it does require a slight tweak in order increment by 1 for each attacking queen.

A sample transition is not included, as it is far more verbose in this case, and the above enumeration is likely to provide more clarity.

G. Property Specification

The primary property specification we utilize is provided by equation 5. In natural language this specification reads, "with what probability is it eventually the case that overall attack equals false", or more naturally "with what probability is a solution eventually found." Since experiments are run over a value range, this probability returned by PRISM after each transition (iteration) is the probability that a solution will be found by that number of transitions (iterations). In equation 5, Overall_Attack refers to a formula which can determine if any queen is under attack from a column or diagonal (rows are not considered as we never have queens on the same row). Unfortunately, we have not found a way to utilize PRISM's renaming functionality to avoid explicitly creating this complete formula. An example of what this formula looks like for 4 queens is included below. Here Overall_Attack refers to a formula which can determine if any queen is under attack from a column or diagonal (rows are not considered as we never have queens on the same row). Unfortunately, we have not found a way to utilize PRISM's renaming functionality to avoid explicitly creating this complete formula. Equation 6 provides an example of what this formula looks like for 4 queens.

$$P =?[F!Overall\_Attack] \tag{5}$$

$$\begin{gathered}\text{formula } overall\_attack = \\ ((q1x - q2x) = (q1y - q2y) | (q1x - q2x) = -(q1y - q2y)) | \\ (q1x = q2x) | \\ ((q1x - q3x) = (q1y - q3y) | (q1x - q3x) = -(q1y - q3y)) | \\ (q1x = q3x) | \\ ((q1x - q4x) = (q1y - q4y) | (q1x - q4x) = -(q1y - q4y)) | \\ (q1x = q4x) | \\ ((q2x - q3x) = (q2y - q3y) | (q2x - q3x) = -(q2y - q3y)) | \\ (q2x = q3x) | \\ ((q2x - q4x) = (q2y - q4y) | (q2x - q4x) = -(q2y - q4y)) | \\ (q2x = q4x) | \\ ((q3x - q4x) = (q3y - q4y) | (q3x - q4x) = -(q3y - q4y)) | \\ (q3x = q4x);\end{gathered} \tag{6}$$

H. PRISM Model Files

The fully specified files used in this work have been may available to the public on GitHub under an MIT license. See appendix A for more information.

IV. Trial Experiments

A. PRISM Experiments

Table I shows 6 trials chosen to be analyzed with PRISM experiments. Q represents the number of queens (and consequently the length and width of the board) and N represents the number of transitions (iterations) that the experiment analyzes. Trials 1 and 4 analyze the random column move algorithm up to 450 transitions (iterations) for both the 4 queens case and 5 queens case, respectively, trials 2 and 5 analyze the random column swap algorithm up to 450 transitions (iterations) for both the 4 queens case and 5 queens case, respectively, and trials 3 and 6 analyze the random column swap algorithm up to 450 transitions (iterations) for both the 4 queens case and 5 queens case, respectively. The experiments consist of constructing the full set of DTMCs from the stochastic transition systems and finding the probability from our specification (see III-G) as a function of the number of transitions (iterations) taken.

TABLE I: trials

Trial	Stochastic Algorithm	Q	N	Step Size
1	Random Permutations	4	450	10
2	Random Column Swaps	4	450	10
3	Simulated Annealing	4	50	5
4	Random Permutations	5	450	10
5	Random Column Swaps	5	450	10
6	Simulated Annealing	5	50	5

V. Experiment Results

The results of the 6 experiments are provided in tables II and III. Each table has 3 trials; one for each stochastic algorithm. Tables II and III provide the number of iterations required to reach a 99% probability that a solution has been found for cases of 4 queens and 5 queens, respectively. Graphs describing the probability that a solution has been found based on the number of transitions (iterations) are provide for each case in Figures 6, 7, 8, 9, 10, and 11.

A. 4 Queens

As seen in figure 6, for the case of 4 queens, the probability that a solution is found for the random column move algorithm follows a smooth hyperbola reaching approximately 99% after 400 iterations. Figure 7 shows that the probability that a solution is found for the random column swap algorithm follows a smooth hyperbola reaching approximately 99% after 50 iterations. This is a significant improvement when compared to the random column move algorithm. Figure 11 shows that the probability that a solution is found for the simulated annealing algorithm follows smooth hyperbola reaching approximately 99% after 45 iterations. This is a slight improvement over the random column swap algorithm (and consequently a major improvement of the random column move algorithm).

B. 5 Queens

Figure 9 shows that the probability that a solution is found for the random column move algorithm follows a smooth hyperbola reaching approximately 99% after 450 iterations. The result is closer to a linear model than it was in the 4 queens case; however, it takes approximately 50 more transitions to reach approximately 99% probability. Figure 10 shows that the probability that a solution is found for the random column move algorithm follows a smooth hyperbola

TABLE II: Experiment Results: 4 Queens

Trial	Stochastic Algorithm	99% Probability Property Satisfied
1	Random Permutations	400
2	Random Column Swaps	50
3	Simulated Annealing	45

TABLE III: Experiment Results: 5 Queens

Trial	Stochastic Algorithm	99% Probability Property Satisfied
4	Random Permutations	450
5	Random Column Swaps	49
6	Simulated Annealing	45

reaching approximately 99% after 49 iterations. Again, this is a significant improvement when compared to the random column algorithm; however, it is very similar to the 4 queens case in terms of the probability that a solution is found. Figure 11 shows that the probability that a solution is found for the simulated annealing algorithm follows a smooth hyperbola reaching approximately 99% after 45 iterations. This is a slight improvement over the random column swap algorithm (and again, a major improvement over the random column move algorithm); however, there is relatively little change between the 4 and 5 queens versions of the simulated annealing algorithm.

C. Sanity Checks

A sanity Checks was performed by testing the property shown in equation 7. In natural language the specification states, the probability is .99 that no queens are under attack in every element of a given trace. Since the queens are initially under attack, this property should return false, and of course did.

$$P = .99[G!Overall\_Attack] \tag{7}$$

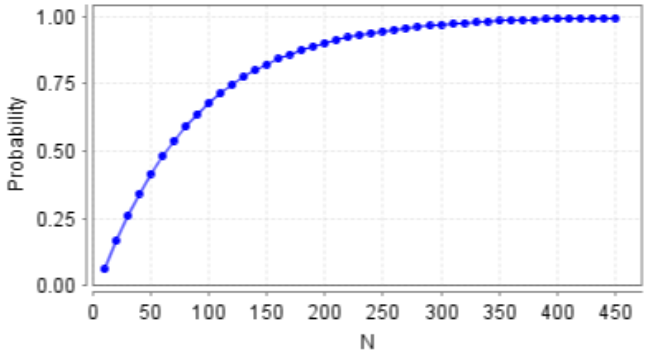

Fig. 6: Random Column Move with 4 Queens

VI. Challenges

A. State-Space Explosion

There are multiple challenges when attempting to provide probabilistic guarantees about a stochastic algorithms. First, there is the classical problem of the state-space explosion. The

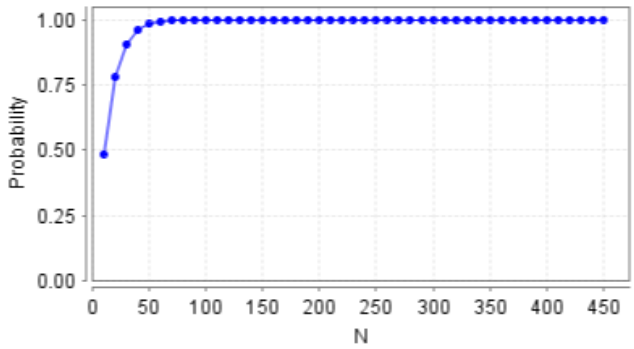

Fig. 7: Column Swap with 4 Queens

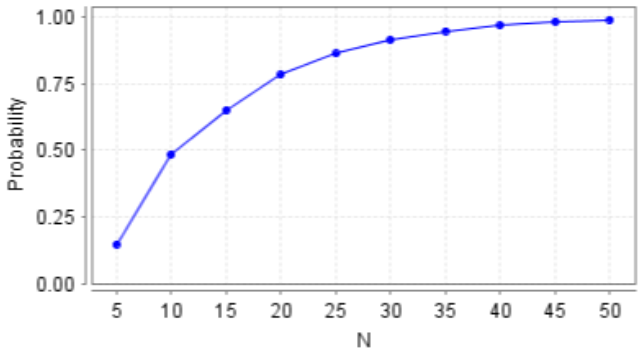

Fig. 8: Simulated Annealing with 4 Queens

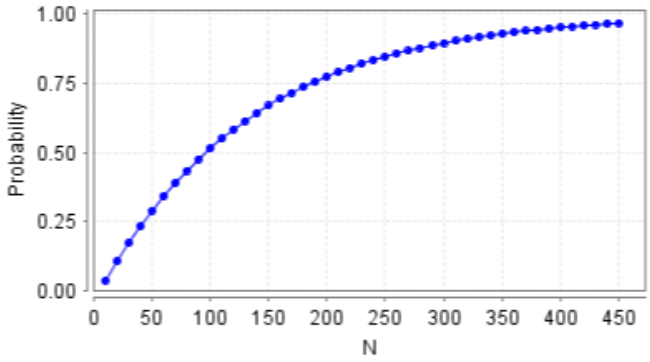

Fig. 9: Random Column Move with 5 Queens

list below describes the state-space explosion as the number of queens, Q, is increased.

1) Random Column Permutation
   a) 4^4 x I (4^4 x 450 = 115,200)
   b) 5^5 x I (5^5 x 450 = 1,406,250)
   c) 100^{100} x I (100^{100} x 450 = 4.5 x 10^{202})
   d) Q^Q x I
2) Random Column Swap
   a) 4^4 x I (4^4 x 450 = 115,200)
   b) 5^5 x I (5^5 x 450 = 1,406,250)
   c) 100^{100} x I (100^{100} x 450 = 4.5 x 10^{202})
   d) Q^Q x I

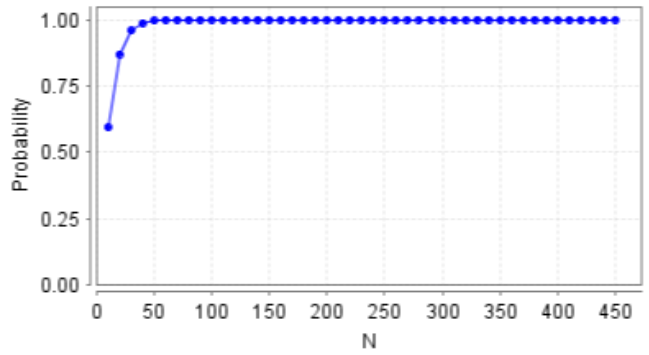

Fig. 10: Column Swap with 5 Queens

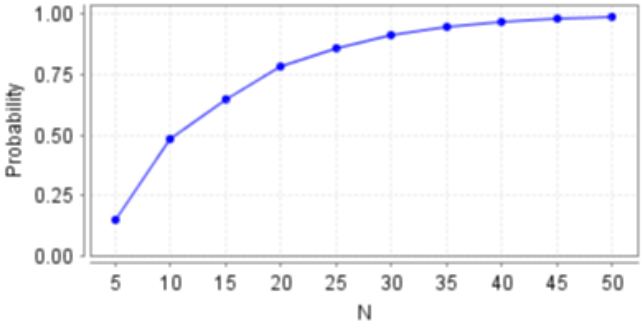

Fig. 11: Simulated Annealing with 5 Queens

3) Simulated Annealing

a) 100 x Q^Q x I

Here the I represents the maximum number of iterations for which the algorithm can run. This inclusion is unfortunate as it yields a vastly increased number of states which PRISM must consider, where in reality many of those states are 'identical' in some sense. Unfortunately, we were unable to find a method within PRISM which would allow us to compute the probability of something within a certain number of steps without introducing artificial deadlock states which cause these I terms to be present.

The 100 is present in Simulated Annealing, as, unfortunately, the probability is a state variable, and therefore further increases the number of states PRISM must track. A possible improvement on this would be to tie the probability to the current transition number, which could possibly lead to some reductions.

Note that the number of states is exponential with respect to the number of Queens. Some of those states can never be reached, but it is still the case that if we have Q queens, then this means we have Q state variables for those queens, each with Q possible values, leading to Q^Q possible combinations between them.

B. Search/Solution Space Explosion

Unfortunately, PRISM has a hard time dealing with the N-Queens problem, though this is for a fairly clear reason. Consider the number of possible placements of N queens on an N by N board. It is not difficult to see that it is $(N^2)(N^2-1)(N^2-2)...(N^2-(N-1))$ which is equivalent to $(N^2!)/(N^2-N)$ which is a very fast growth rate. Fortunately, we do not actually consider every possible board.

Consider, first, the Perturbation algorithms. While queens are allowed to be in the same columns, they are not allowed to be in the same row. Therefore, the number of possible queen placements during this algorithm is given by N^N, as there are N choices for each queen.

Next consider the Swap algorithms. In addition to the row restriction, there are now column restrictions. Thus there are N choices for the first queen, $(N-1)$ for the second, etc, yielding $N!$ possible placements in these algorithms.

Now note that each of these reductions is strictly beneficial for us, as all solutions to the N-Queens problem require that all queens be on different rows and columns, so the placements we exclude with each algorithm are only of the sort which could not be an actual solution. Unfortunately, $N!$ is still a very fast growth rate.

C. DTMCs Explosion

The biggest challenge we face with this stochastic analysis is the explosion of the DTMCs graphs as the number of transitions increases. Since each stochastic transition, if enabled, performs one of several actions based on a probability, the number of possible transitions from time 0 grows rapidly as as the number of transitions (iterations) increases. Consider the transition system for the random column move algorithm with N queens. There are N transitions, each with $N-1$ possible updates based on a $\frac{1}{N-1}$ probability. For the first transition there are $N(N-1)$ number of possible outcomes, for the second transition there are $(N(N-1))^2$ possible outcomes, and so on. Assuming I to be the number of transitions, there are $(N(N-1))^I$ possible outcomes after I transitions (not including repeat states). Though the computational graph can be reduced as repeat states occur using reduced-order binary decision diagrams (ROBDDs), the scalability problem still presents an issue.

D. PRISM Memory Limitations

All of the aforementioned challenges reveal the memory limitations of PRISM. Higher number of queens (greater than 5) are not able to be evaluated due exponentially increased time-to-completion and out-of-memory errors.

E. Future Work

It would be interesting to implement a greedy swap algorithm that only takes a swap if it reduces the total number of attacked queens on the board. It would also be interesting to increase the memory limitations within the PRISM source code so that it could handle larger N Queens problems. Last, it would be beneficial to debug PRISMs reward functionality which allows PRISM to perform state space reductions.

VII. Conclusion

A. Summary

We presented a probabilistic model checking approach using PRISM to verify a PCTL property of three stochastic algorithms used to solve N-Queens; random column swap, random column move, and simulated annealing. The PCTL property that we evaluated was the probability that it is eventually the case that no queen is under attack (solution state). We ran 6 trial experiments in PRISM to evaluate this property and return the probability that the respective algorithm will reach a solution state as a function of the number of transitions taken by the underlying stochastic transition systems. Trials 1 through 3 evaluated the 4-queens case, and trials 4 through 6 evaluated the 5-queens case. Trials 1 and 4 evaluated the random column move algorithm and revealed an approximately 99% probability that a solution is found after 400 and 450 transitions, respectively. Trials 2 and 5 evaluated the random column swap algorithm and revealed an approximately 99% probability that a solution is found after 50 and 49 transitions, respectively. Trials 3 and 6 evaluated the simulated annealing algorithm and revealed an approximately 99% probability that a solution is found after 45 transitions each. We concluded by discussing the challenges associated with probabilistic model checking; namely, the state-space, search/solution space, and DTMCs explosion problems, and by provide suggestions for future work.

Appendix

A. Software

The model files and Python meta-scripts for this work have been made publicly available on GitHub under an MIT license at the following location https://github.com/rpicard92/prism-stochastic-n-queens-analysis.

B. PRISM Figures

References

[1] "Randomised dining philosophers." [Online]. Available: https://www.prismmodelchecker.org/casestudies/phil.php

[2] D. Lehmann and M. O. Rabin, "On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem," in Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '81. New York, NY, USA: ACM, 1981, pp. 133–138. [Online]. Available: http://doi.acm.org/10.1145/567532.567547

[3] "Board editor." [Online]. Available: https://lichess.org/

[4] Y. Baror, "Visualgos: N queens." [Online]. Available: http://yuval.bar-or.org/index.php?item=9

[5] D. Soni, "Introduction to markov chains." [Online]. Available: https://towardsdatascience.com/introduction-to-markov-chains-50da3645a50d

[6] D. Parker, "Discrete-time markov chains." [Online]. Available: http://www.prismmodelchecker.org/lectures/pmc/03-dtmcs.pdf

[7] ——, "Probabilistic temporal logics." [Online]. Available: http://www.prismmodelchecker.org/lectures/pmc/04-prob\%20logics.pdf

[8] "Prism." [Online]. Available: http://www.prismmodelchecker.org/

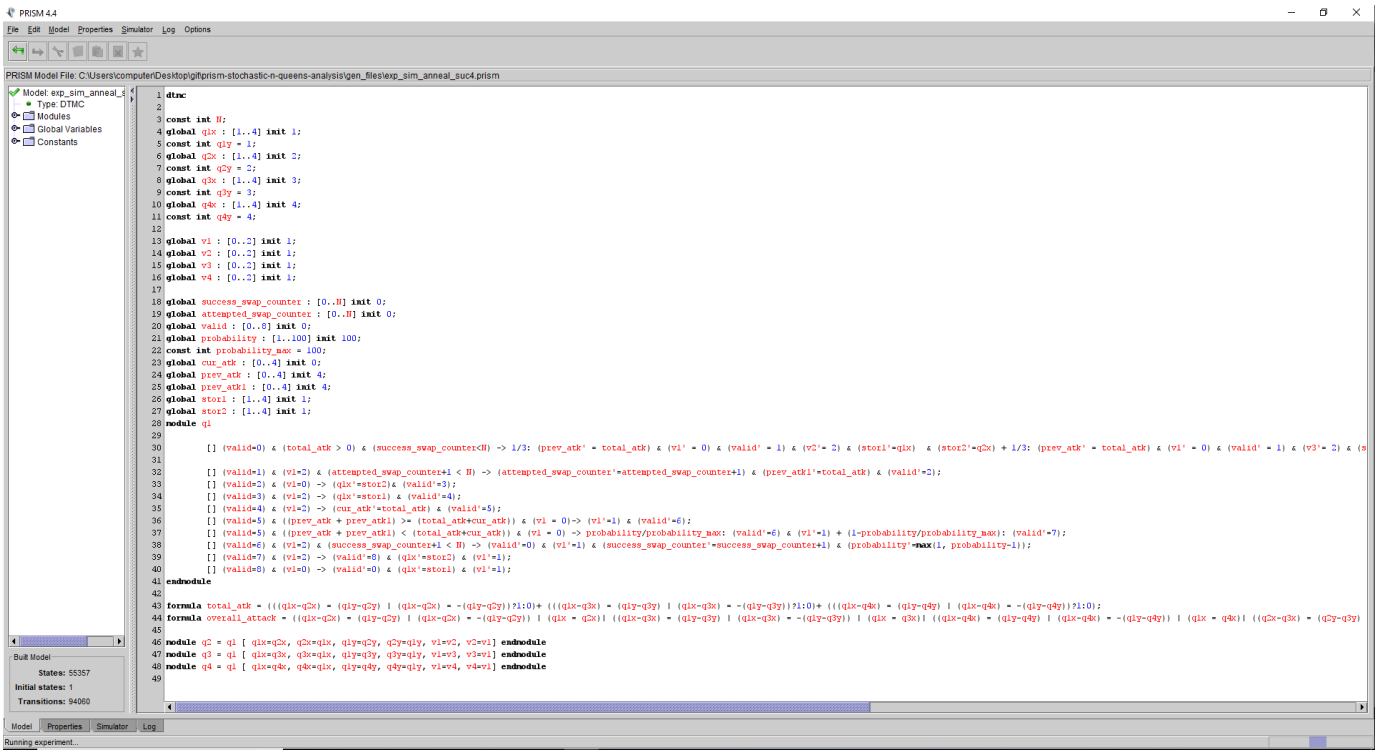

Fig. 12: PRISM GUI: Model Specification View

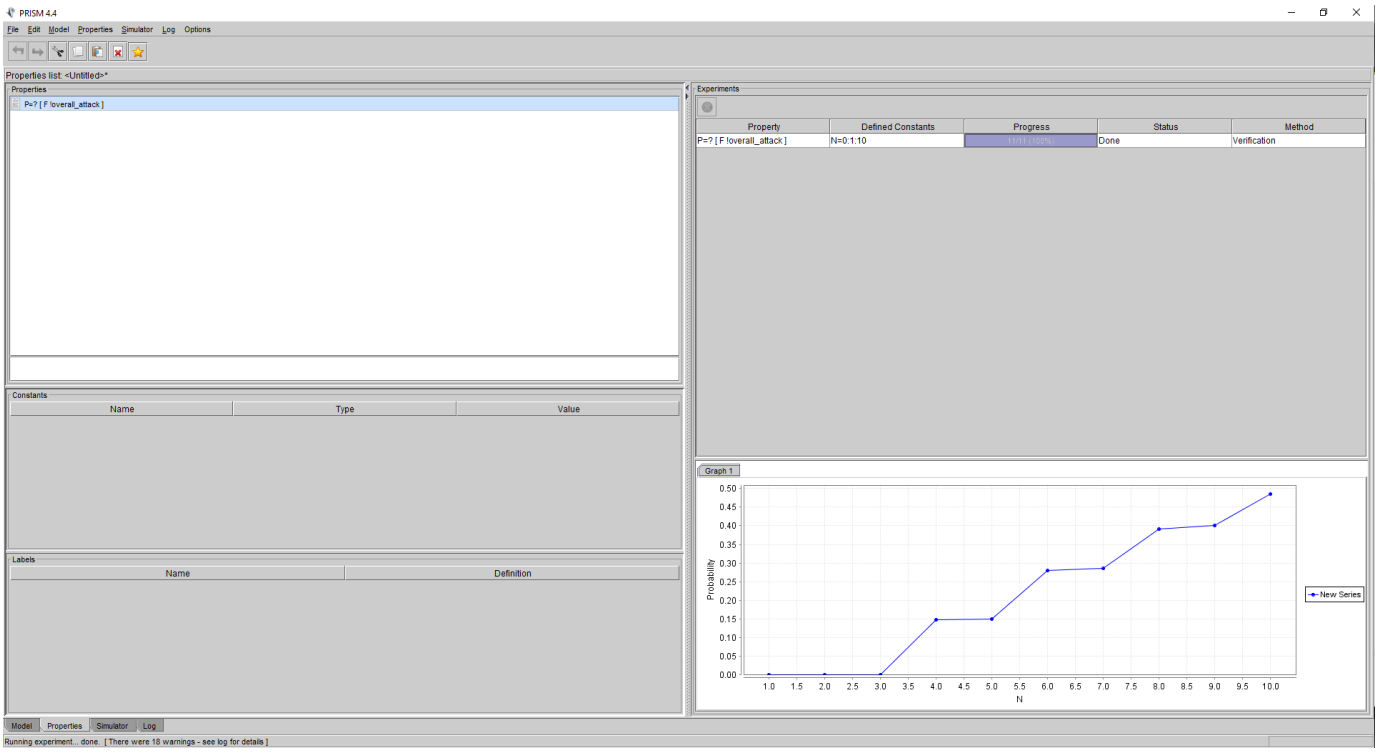

Fig. 13: PRISM GUI: Proper Specification and Experiment View