Searching for reachability property in complex software systems specified through graph transformations using deep reinforcement learning

Mohammad Javad Mehrabi Vahid Rafe

Department of Computer Engineering, Faculty of Engineering, Arak University, Arak 38156-8-8349, Iran

mj.mehrabi96@gmail.com v-rafe@araku.ac.ir

Abstract

Today, model checking is one of the essential techniques in the verification of software systems. This technique can verify some properties such as reachability in which the entire state space is searched to find the desired state. However, model checking may lead to the state space explosion problem in which all states cannot be generated due to the exponential resource usage. Although the results of recent model checking approaches are promising, there is still room for improvement in terms of accuracy and the number of explored states. In this paper, using deep reinforcement learning and two neural networks, we propose an approach to increase the accuracy of the generated witnesses and reduce the use of hardware resources. In this approach, at first, an agent starts to explore the state space without any knowledge and gradually identifies the proper and improper actions by receiving different rewards/penalties from the environment to achieve the goal. Once the dataset is fulfilled with the agent's experiences, two neural networks evaluate the quality of each operation in each state, and afterwards, the best action is selected. The significant difficulties and challenges in the implementation are encoding the states, feature engineering, feature selection, reward engineering, handling invalid actions, and configuring the neural network. Finally, The proposed approach has been implemented in the Groove toolset, and as a result, in most of the case studies, it overcame the problem of state space explosion. Also, this approach outperforms the existing solutions in terms of generating shorter witnesses and exploring fewer states. On average, The proposed approach is nearly 400% better than other approaches in exploring fewer states and 300% better than the others in generating shorter witnesses. Also, on average, the proposed approach is 37% more accurate than the others in terms of finding the goals state.

Keywords: Reachability, Verification, Deep Reinforcement Learning, Model Checking, State, State Space Explosion, Graph, Graph Transformation System, Artificial Intelligence, State Space, Witness, Initial State, Goal State, Terminal State, Node, Edge, Rule, Transition, Exploration, Search, Deadlock

1. Introduction

Model-checking is one of the formal approaches in the verification of software systems. This technique is used to verify (or refuse) a desired property, such as reachability in software systems. In order To verify a reachability property, the model checker should generate and check the entire state space to find the desired state. If the desired state (i.e. goal state) is found, a path from the initial state to the goal state will be shown to the user. Such a path is called a witness. One of the limitations of model checking is the state space explosion problem [1]. Due to the nature of model checking, it is needed to check all states to find the desired state. In complex systems which have a very large or infinite state space, the problem of state space explosion will occur in which the state space cannot be explored entirely [2]. In the past years, several methods such as Genetic Algorithm [3], A* [4], and Bayesian Optimization Algorithm [5] have been proposed to explore the state space intelligently to overcome the state space explosion problem. However, they have some limitations, and there is still room for improvement. For example, in some case studies, their accuracy is low. The length of the generated witnesses and the number of explored states is not optimized. This paper proposes an approach based on deep reinforcement learning to cover these limitations.

Like supervised and unsupervised learning, reinforcement learning is one of the machine learning methods and is called real artificial intelligence. In the reinforcement learning method, an agent first enters the state space without any knowledge. After acquiring information about the environment, the agent learns how to behave to reach the goal. This information includes rewards and penalties that the environment will give to the agent along the way.

The proposed approach combines deep reinforcement learning with model checking and uses the graph transformation system (GTS) [6] to model the system. In GTS based systems, after applying a rule, the system goes to another state and therefore, rules can be considered as an agent's actions. Here, the model checker that applies the rule plays the agent's role. After applying a rule, differences between the current state and the goal state can be calculated and then considered as a reward or penalty which is given to the agent. The agent remembers the applied rule and its results and stores them in the memory. When the number of experiences is exceeded from a threshold value, a batch of experiences randomly is selected and then is used to train the neural networks in which the main neural network chooses the next best rule, and the target neural network calculates the q value of the selected rule. Meanwhile, the main neural network predicts the next best rule, and the agent will apply it. The main contributions of the proposed approach are as follows:

1) Finding shorter witnesses in comparison to existing approaches,
2) Finding the goal states intelligently in terms of exploring fewer states than other approaches,
3) Better performance in terms of accuracy in comparison with the existing approaches,
4) Adopting the machine learning algorithms in search-based model checking

In this paper, the GROOVE toolset [7] is used to implement the proposed approach. In the implementation phase, we overcame some major difficulties. First, overcame the challenge of feature engineering and feature

selection for encoding the states. Second, reward engineering for calculating the performance of the selected rule by calculating the difference between the current state and the goal state. Also, for each case study, we offer a dedicated reward function. Third, we overcame the challenge of invalid actions by limiting the number of applicable rules. Finally, we overcame the challenge of configuring the neural network to reach a responsible one for most case studies by testing multiple setups of neural networks.
To evaluate the efficiency of the proposed approach, we compare the obtained results on different benchmarks with the existing state-of-the-art techniques.

The rest of the paper is organized as follows: In Section 2, related works are presented. Section 3 briefly introduces the necessary background, such as reinforcement learning, model checking and GTS. In Section 4, it is presented that how deep reinforcement learning will be integrated with model checking. Experimental results are presented in Section 5. Also, at the end of this section, the advantages and limitations of the proposed approach have been discussed. Finally, we conclude the paper in Section 6 and propose directions for future works.

2. Related Work

In recent studies, some approaches have been proposed to explore a portion of the state space instead of the whole. By using these approaches, more states can be explored intelligently; therefore, the state space explosion problem can be delayed. These approaches can be divided into three categories described in the rest of the section. Also, the recent works in the field of integration of model checking and reinforcement learning will be discussed.

2.1. Using simple heuristic search algorithms

Some well-known search algorithms, such as A* and IDA*, are included in this category. In [4], A* algorithm is used on graph transformation system to solve the planning problems. Although one of the drawbacks of this method is defining the weights manually, this approach performs much better than a simple approach like DFS. In [8], an approach called Depth-first Heuristic Search is implemented inside Java Path Finder. This approach explores the state space just like the depth-first search but has some differences. Some paths that are not likely to satisfy the desired property will be ignored.
Even though most of these approaches have a simple structure and run fast, their results are not very accurate.

2.2. Using meta-heuristic and evolutionary algorithms

Yousefian et al. [3] proposed a model checking technique based on the Genetic Algorithm. In this approach, every path is called a chromosome. In every level of exploration, the proposed algorithm detects which paths and states should be explored at the next level.

In [9], Greedy and BAPSO were proposed. The last one is the combination of the Bat and PSO algorithms. The authors concluded that BAPSO could handle the state space explosion and perform better than the BAT and PSO.

In [10], the authors proposed an approach that is based on the ant colony optimization algorithm for analyzing reachability properties in systems specified formally through a graph transformation system.

In [11], the authors first proposed an evolutionary algorithm to check reachability properties in software systems specified formally through graph transformations. Finally, to improve the accuracy and convergence speed of the proposed approach, the authors employed the Bayesian Optimization Algorithm (BOA) to propose another approach.

2.3. Using knowledge discovery and machine learning techniques

In [12], the authors proposed two approaches that are called Learning by Data Mining and Learning a Bayesian Network and both of them are used to satisfy the reachability property. In both approaches, at first, the BFS algorithm is used to explore a portion of state space to obtain a dataset of knowledge discovery. Afterwards, the rest of the state space will be explored efficiently with the help of the obtained dataset.

In [5], Pira et al. proposed an approach based on Bayesian Optimization Algorithm to detect deadlock in graph transformation systems. This approach is based on the rule dependency graph. Rule r_2 depends on r_1 if r_1 adds/removes at least one node or one edge that its existence/absence in s is necessary for applying the rule r_2. This approach generates better accuracy and depth of satisfied property results than the Genetic and PSO algorithms. However, the process of learning and saving the table of conditional probabilities is one of the limitations in this approach. Moreover, if there is no dependency between rules, this approach will not work properly.

Pira has proposed an AI planning approach [13] that is called plnBOA and is based on refinement technique and Bayesian optimization algorithm. In this approach, the author removes the defined percentage of nodes from each grouped goal state goal to make a refined goal state. After that, the author deploys the BOA algorithm to solve the planning problem for the refined goal state. Eventually, the author uses the last BN learned in BOA to solve the planning problem for the main goal state. The advantages of this approach are faster execution speed, shorter length of generated plans, fewer explored states, and higher accuracy in comparison with the others. However, plnBOA has some limitations such as requiring a type graph while other approaches do not need that. Also, it takes much time to generate the refined goal state, and the learning of BNs is time-consuming.

Pira et al. [14] have proposed the EMCDM approach that uses data mining to explore effectively in the state space. In fact, in the beginning, the desired property is checked on a smaller model. Afterwards, the knowledge is extracted. This approach has the benefit in terms of speed than other approaches. However, generating a proper smaller model is challenging for complex systems.

Partabian et al. [15] have proposed an approach that generates a small model consistent with the main model. Then it explores the state space entirely to find the goal states and labels the paths which start from the initial state and lead to a goal state. Then using the ensemble classification technique, the necessary knowledge is extracted from these paths to explore the state space of the bigger model intelligently.

In [16], the authors present a method that employs machine learning techniques without exploring the whole state space to predict temporal properties of trajectories in systems based on graph transmission system. The proposed method is implemented in Groove.

Pira in [17] proposed a two-phase approach. In the first phase, the beam-search algorithm explores the state space to a specific number of states. In case of failure of the phase, the second phase starts. In the second phase, a Markov chain (MC) is estimated to capture dependencies between the sequence of applied rules in the state space explored by the beam-search algorithm. The MC is then employed to intelligently explore the remainder of the state space.

2.4. Integration of reinforcement learning and complex systems

In [18], the authors proposed an approach to detect safety property that guarantees the excellent behaviour of the agent in the environment using a technique called Shield. In this approach, to limit the agent's behaviour, the time limitation is applied.

In [19], an approach has been proposed, which combines model checking with reinforcement learning. This approach increases the accuracy with the help of learning and guarantees the safety of decisions made by reinforcement learning with the help of model checking. In all of the above approaches, model checking is just used to monitor the agent and detect safety property. Also, these approaches are not based on graph transformation systems.

Liu et al. [20] proposed an approach for complex system control called parallel reinforcement learning. This approach addresses some issues like data inefficiency, data dependency and distribution and lack of generalization capability to new goals. The authors also proposed a bidirectional long short-term memory based deep reinforcement network (BiLSTM-DRN) to approximate the action-value function to train data in the large action and state space.

3. Background

In this section, required background such as model checking, GTS, reinforcement learning and Double Deep Q-Network are reviewed briefly.

3.1. Model Checking

Model-checking is one of the formal verification techniques based on system modelling and is generally used to ensure the system's reliability. By using model checking, we can determine if there is an error in the software system. While giving complete confidence and trust to the developers, model checking has its drawbacks. Due to checking all states in model checking to find the desired property, in large or infinite state space, a technical limitation occurs that is called state space explosion. Several properties, like reachability, can be checked using model checking. In Reachability property, it will be checked if something special is going to happen. Reachability property determines whether or not the existence of a user-desired state is called the goal state in the whole system.

Assuming that a property g is a particular configuration of the system, it is reachable if a state exists in the model state space in which the property g occurs (Fig. 1). In this paper, GTS formalism is used to model the systems. Hence, each state in the model state space is a graph that is typed over the type graph of the large model. Also, the property g is a graph by which a desirable and specific configuration of the given system is described.

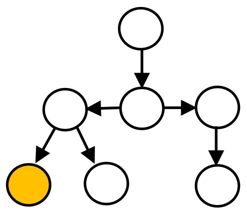


Fig. 1. Reachability property.

3.2. Graph Transformation System

Graph Transformation System (GTS) is a language for formal modelling that describes states and dynamics of a software system using graphs and graph transitions [6]. GTS has several benefits. The most important of these benefits is to display the states as a graph. Because graphs are visual and explicit, it removes useless details [21]. The states of a system are displayed as graphs, and changes can be applied to them by using rules. A GTS is a triple (R, HG, TG) in which R a is a set of graph transformation rules, HG is a Host Graph, and TG is a Type Graph. A Type graph is a graph that represents the entities of the system and the relationship among them. In this graph, every entity is specified using a type node, and the relationships between them are represented by an edge. Type graph is a combination of nodes(N), edges(E), $src: E \rightarrow N$ and $tgt: E \rightarrow N$ for declarations of any type of edges. *src* means source and *tgt* means the target of edges. Host graph shows the initial state of a system and should be an instance of the type graph.

Generally, rules are triple (RHS, LHS, NAC). LHS (left-hand side) contains preconditions to apply a rule, RHS (right-hand side) is the result of applying the rule on LHS and NAC is negative application conditions. Nodes and edges that are present in LHS but not in RHS will be removed from the new state after applying a rule. Also, nodes and edges that are present in RHS but not in LHS will be added to the new state after applying the rule.

There are several toolsets, such as AGG [22], ATOM3 [23], VIATRA3 [24] and GROOVE [7]. This paper uses the GROOVE toolset because it performs model checking by producing the model's state space. We use the dining philosopher problem as an example of a system modelled by GTS in the GROOVE toolset. In this problem, there are several philosophers, and also there is a fork between each pair of them. In the beginning, the philosophers are thinking, and after a while, they get hungry. A hungry philosopher should take left and right forks, respectively, to eat. Finally, after eating, they release the forks and go again to the thinking state. The type graph of this problem is presented in Fig. 2(a). As it is shown, philosopher (*n0*) can have two forks as a maximum (i.e., *n1* and *n2)*. The left fork (*n1*) with edge (*e1*) and the right fork with edge (*e2*) is connected to the *n0*.

Moreover, the state of philosopher (*n0*) can be one of these states: eating (*eat*), taking the left fork (*hasLeft*), taking the right fork (*hasRight*), going hungry (*hungry*) and thinking (*think*). Host graph (start state) with two philosophers are shown in Fig. 2(b). As it is shown, two philosophers are labelled with *n0* and *n1*. The philosopher (*n0*) is connected to his right fork *n2* and left fork *n3* with *e0* and *e1* edges, respectively. Also, philosopher *n1* is connected to his left and right fork (*n2*, *n3*) with *e2* and *e3* edges. As mentioned above, one of the rules in this problem is to take the right fork (e.g., taking fork *n2* by philosopher *n0*). LHS of this rule is equal to the state in which philosopher *n0* took the left fork. NAC, in this rule, is a state in which the right fork is not taken by another philosopher (e.g., *n1*). Finally, RHS (final graph) is equal to taking the right fork by philosopher *n0* and after that going to eat state (Fig. 2(c)). As it is shown, the black edges are the edge that should remain in RHS after applying the rule. Green edges are the edges that will be added to RHS. Red edges are NACs, and blue dashed edges are the edges that exist in LHS but should remove in RHS.

Definition 1 (explored states): The states that are generated in state space and stored in memory. Finding the goal state with fewer states exploration cause to save the memory and also reveal the accuracy of proposed approach exploration.

Definition 2 (witness): A witness specifies an initial and finite path $S_0R_0S_1R_1S_2R_2\ldots S_G$ such that the desired property is satisfied in the state S_G (e.g. goal state). *S* and *R* are states and applied rules, respectively. S_0 is the initial state. GTS generates a witness if the goal state is found in the state space. In fact, An algorithm that finds shorter witnesses is managed to find the goals in lower depth than other approaches. It means the algorithm can solve the problem with less action than other algorithms.

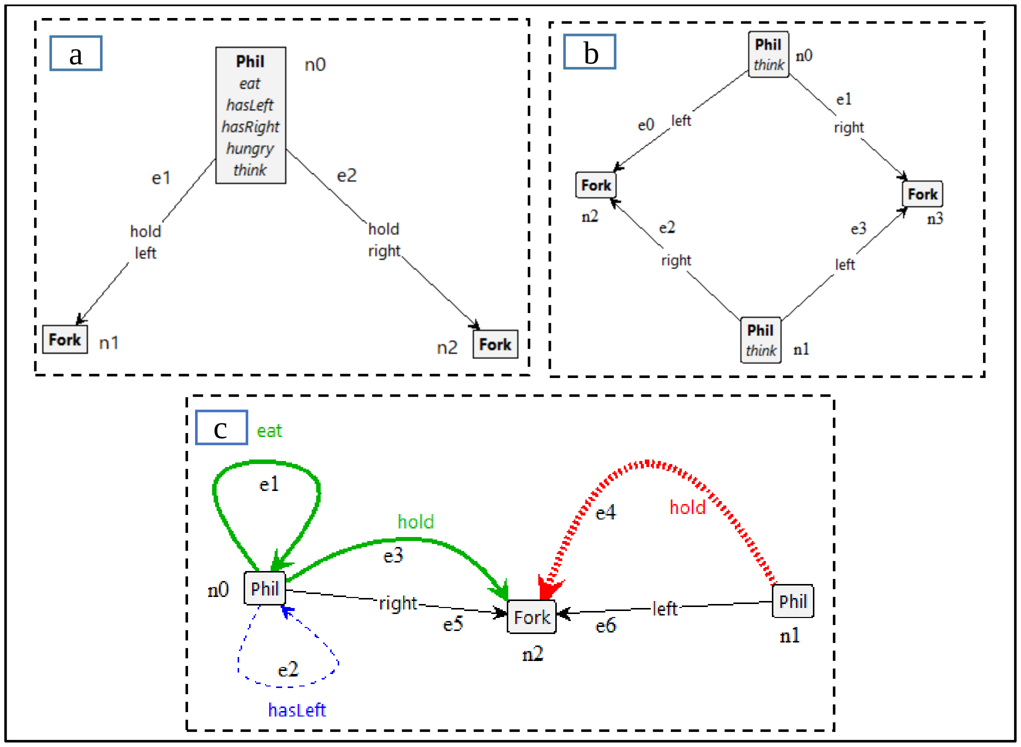


Fig. 2. A GTS-based model of the dining philosophers problem designed in the GROOVE toolset.

3.3. Reinforcement Learning

Reinforcement learning, along with supervised and unsupervised learning, is one of the subsets of machine learning. Also, this learning method is called real artificial intelligence because, at first, the agent does not have any knowledge about the environment, and after getting some information, it learns how to act to achieve the goal. This learning is the result of getting rewards and penalties by the agent after performing any action in the environment. To learn how to achieve the goal, the agent needs to interact with the domain of the problem. In this way, the dynamics and statics of the system can be learned to find which actions are the best ones to achieve the goal [25]. A conceptual model for reinforcement learning is shown in Fig. 3. In this figure, the agent receives a state from the environment in each timestamp and performs an available action. In the next timestamp, according to the agent's action, it will receive the reward from the environment and take one step into the new state.

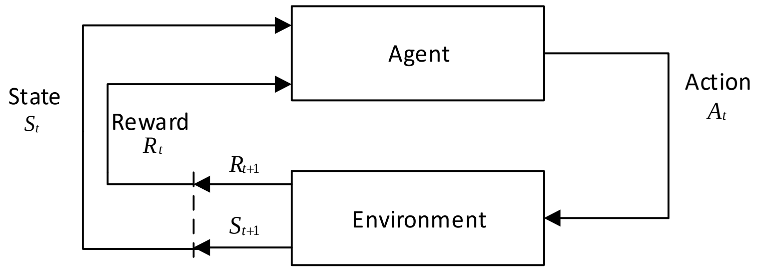


Fig. 3. A conceptual model for reinforcement learning [25].

In classic reinforcement learning algorithms like Q-Leaning [25] and SARSA [25], tables are used to save the knowledge about the environment. This way, one row and one column for every action and state are allocated.

Definition 3 (experience): The (S, A, S′, R) tuple is known as an experience for the agent [26] in which:

- S: Current state in which the agent is placed.
- A: Selected action (i.e., the selected rule in model checking).
- S′: A state that the agent goes to by selecting action A.
- R: A reward or penalty that the agent receives after going from state S to state S′ through action A.

3.4. Deep Reinforcement Learning Using Double Deep Q-Network

In simple reinforcement learning, due to the table's overhead, especially in large state spaces, it is impossible to store all actions and states in a table. Also, at the beginning of the exploration, many unknown actions and states will be missed in the initial table. To overcome this problem, with the help of a neural network, older experiences of the agent are used to predict the output of the value function [27]. In this way, the Q-values table is removed, and the neural network is used to predict the Q-values. Double Deep Q-Network (DDQN) [28] is the combination of Q-Learning and neural networks. In this algorithm, It just needs to give the current state to the system and predict the Q-value for each available action in that state. After this prediction, the action with that highest Q-Value is selected, and the agent performs that.

In this approach, two neural networks are used to overcome the overestimation problem. The main neural network is used to choose the best action with the highest Q-Value, and after that, with the usage of the target network, the target value of the selected action will be calculated. The rules are considered as actions in the GTS, and the Q-Values are calculated for each rule. The rule with the highest Q-Value will be applied to the current state.

Definition 4 (experience replay memory): If the agent only uses its previous experience to learn, an unstable system will emerge in which the agent will always forget its experience. As a result, the agent's experiences need to be stored in a memory that is called experience replay memory [26].

Definition 5 (episodes): the sequence of an agent's movement that starts from the initial state and finishes in the last movement of the agent (i.e., the number of iterations in which an agent stops its movement and starts from the initial state).

Definition 6 (steps): Each movement of an agent. In other words, each time a rule is applied and immediately after that, the current state is changed.

Definition 7 (Epsilon-greedy policy): An exploration policy in which the agent picks the current best option (greedy) most of the time and picks a random option with a small (epsilon) probability sometimes.

4. Our Proposed Approach

In this section, we propose an approach based on Deep Reinforcement Learning to detect reachability in systems specified through GTS. The proposed approach uses the double deep q-network method. The main goal in this section is integrating reinforcement learning with model checking. As it is shown in **Error! Reference source not found.**, at every step, the agent should choose a rule. According to the agent's experiences, the main neural networks helps the agent to choose the next best action. After choosing a rule, the selected rule is applied, and the agent moves to the next state. If the next state is a goal state, then a path from the initial state leading to the goal state will be shown to the user. Otherwise, step 1 will be repeated.

The main challenges are encoding the states, feature engineering, feature selection, reward engineering, handling invalid actions and configuring the neural network to reach a responsible one for most case studies. Also, the number of explored states, length of witness, and time of detection should be optimal in the detection of reachability property.

GROOVE toolset is used to implement the proposed approach (source code is available at https://github.com/mjmehrabi/Groove-RL).

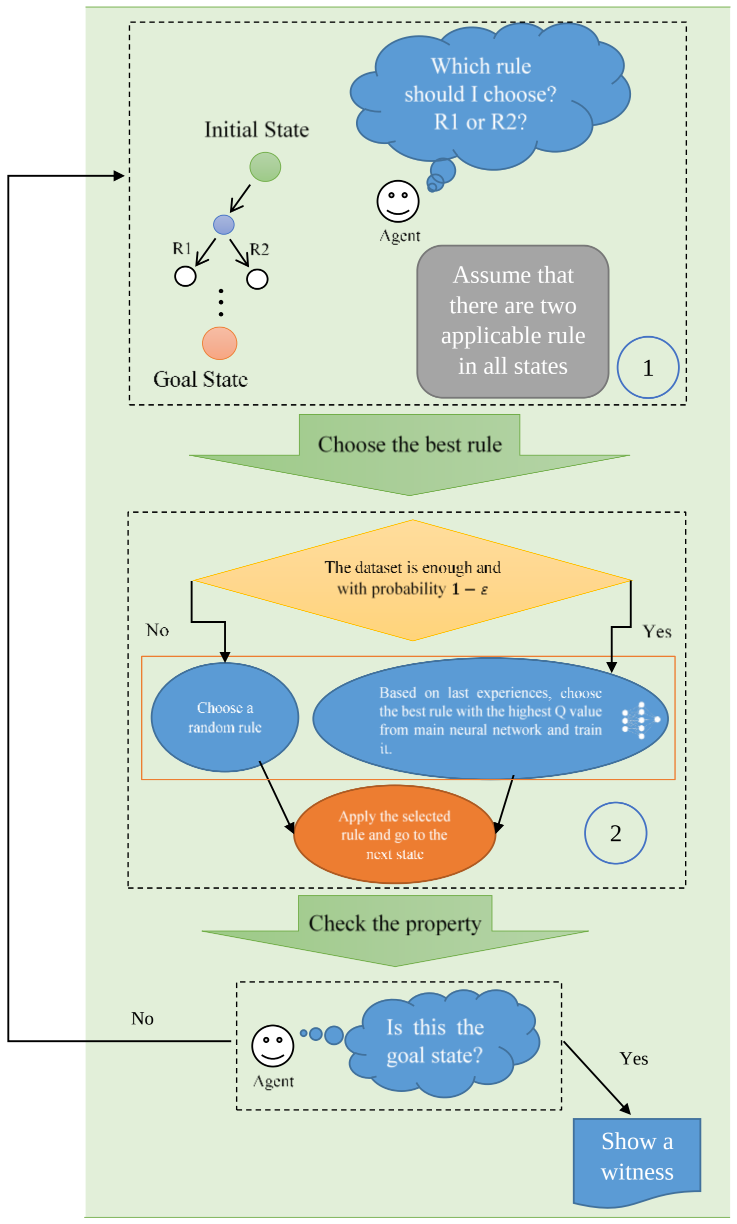

Which rule should I choose? R1 or R2?
Initial State
R1
R2
Goal State
Agent
Assume that there are two applicable rule in all states
1
Choose the best rule
The dataset is enough and with probability $1-\varepsilon$
No
Yes
Choose a random rule
Based on last experiences, choose the best rule with the highest Q value from main neural network and train it.
Apply the selected rule and go to the next state
2
Check the property
No
Is this the goal state?
Agent
Yes
Show a witness

Fig. 4. The general architecture of the proposed approaches.

4.1. Neural Network Configuration

Due to the various parameters in neural networks, an attempt has been made to select a general neural network to solve most problems without any additional configurations. The neural network that is used in this paper is a Multi-Layer Perceptron with one hidden layer with 100 neurons that will be discussed in the next section. The activation layers of the hidden layer and the output layer are a type of ReLU and IDENTITY (linear), respectively. The learning rate of the neural network is adjustable as a hint to the user; 0.1, 0.001 and

0.00001 value is suggested. It should be noted that for high learning rates, a point may be missed when updating the error in the neural network, and the algorithm converges to a local minimum. Also, for low learning rates, the converging may be slow. Thus, the value of 0.001 is set as the default value for the learning rate. Rmsprop is the selected optimizer because it will adapt the learning rate during the operation and adjust it for each parameter.

Due to the variable number of actions that can be performed in the state space and the variability of the size of the state space in different problems, it is not possible to suggest an exact value for the neural network's output. Because the number of outputs in the neural network equals the number of rules that the agent must choose among them, a high constant value of 400 is assumed by default. However, this value is entirely adjustable, and the user can change it according to the number of actions in the model. The input and output details of the neural network are described in detail.

4.2. Reward Engineering

Given that the reinforcement learning algorithm is based on the reward and penalty, the reward function plays a vital role in this system. That is why rewards designing for AI-based systems has become one of the essential topics in this field today. The rewards determine how valuable is this action, and rewards can vary depending on the system. In this paper, two different reward functions are used.

- **Dedicated reward function**

One of the benefits of reinforcement learning is that by only changing the reward function, the agent's behaviour is changed in the environment. In this method, due to the graphical structure of the states, which includes different edges and nodes, the current state must be encoded, and entities in the problem must be identified. The heuristic function must then be written according to the specified entities and the goal state. The term DDQN* refers to the type of implementation of reinforcement learning that uses a specific reward function for each problem.

In Algorithm 1, an example of designing a dedicated reward for blocks world problem is presented. The problem of blocks world is modelled in which there are some red, green, or blue blocks, and the goal is to place the same colour blocks on top of each other. One node is considered as a table block and should be placed under all blocks. In line 1, the algorithm receives two inputs: (1) edges and nodes of the current state, (2) the node that is modelled as a table. The output of this algorithm is the cost value of being in that state, which is, the closer it is to 0, the closer it will be to the goal state. In lines 3 to 9, the blocks will be added to their dedicated lists according to their colours. In lines 10 to 23, if there is at least one block of any colour on the table, it will be considered as the floor block. Otherwise, if there is more than one block on the table, the block that is placed under the most same colour blocks will be chosen as the floor block, and if there are no blocks on the table, the block that is placed under the least number of non-homogeneous blocks, is considered as the floor block. In the last condition, the state's cost is calculated from formula (1).

$$Cost = 1 + n + 5 \quad (1)$$

- 1: The cost of moving the block itself
- n: The number of non-homogeneous blocks
- 5: emphasize the badness of this situation

According to line 25, if there is a block on the arm, one point will be added to the cost. Lines 26 to 29 calculate the cost using formula (2) if a block is not placed on the floor block.

$$Cost = a + b + 2 \quad (2)$$

- **a:** number of all blocks placed on the block
- **b:** number of non-homogeneous blocks on the floor block
- **2:** emphasize the badness of this situation

Algorithm 1 The dedicated reward function for the blocks world problem.

```
1:  Input: edgeset = edge set of state, table = the node that is table
2:  Output: H = heuristic
3:  foreach edge in edgeset
4:      if $label$ = $on$ and target = table then
5:          if label = blue then add to tablesBlueList
6:          else if label = red then add to tablesRedList
7:          else if label = green then add to tablesGreenList
8:      end if
9:  end for
10: foreach colour in Colours
11:     if tablesBlueList.size = 1 then set that as tableBlue
12:     else if tablesBlueList.size > 1 then set tableBlue = that block which has the most same colour on top
13:     else set tableBlue = that block which has the fewer different colours on top and set H += different colour blocks on top + 1 + 5
14:     end if
15:     if tablesRedList.size = 1 then set that as tableRed
16:     else if tablesRedList.size > 1 then set tableRed = that block which has the most same colour on top
17:     else set tableRed = that block which has the fewer different colours on top and set H += different colour blocks on top + 1 + 5
18:     end if
19:     if tablesGreenList.size = 1 then set that as tableGreen
20:     else if tablesGreenList.size > 1 then set tableGreen = that block which has the most same colour on top
21:     else set tableGreen = that block which has the fewer different colours on top and set H += different colour blocks on top + 1 + 5
22:     end if
23: end for
24: foreach edge in edgeset
25:     if $label$ = $holding$ then H += 1
26:     if soure has a blue colour and was not on tableBlue then H = allblockonsource + not the same colour block on table block + 2
```

```
27:         **if** soure has red colour **and** was not on redBlue **then** H = allblockonsource + not the same colour block on table block + 2
28:         **if** soure has red colour **and** was not on tableBlue **then** H = allblockonsource + not the same colour block on table block + 2
29:     **end for**
```

- **General Reward Function**

In this reward function, the similarity between the current state and the goal state is returned as a numerical value. The higher value of this number means significant similarity. In previous approaches, such as the Genetic algorithm and A*, this type of reward function is used to detect reachability property [12]. The term DDQN refers to the type of implementation of reinforcement learning that uses this reward function.

4.3. Feature Engineering and Feature Selection

Feature Engineering is the process of representing a problem domain to make it amenable for learning techniques [29]. Feature selection is the process of obtaining not necessarily an accurate model but a model that is easier to understand and interpret by humans. It helps to remove unrelated data and redundancy [29]. The selection of features is important because if the system is given the wrong features, it will not correctly differentiate between different data (i.e., states) and reduce the algorithm's efficiency. Due to the dynamics of some environments and also the complexity of modelling via graphs and the heaviness of their calculations, especially in large state spaces, it is necessary to look for a feature that, while low and straightforward in computation, increases the learning process. Features are the neural network's inputs that differentiate each data (i.e., states) from another.

In the proposed approach, inputs of the neural network are equal to the sequence of the number of applied rules from the initial state to the current state. Also, the number of neural network's inputs is equal to the maximum allowable steps of the agent in the environment. Assume that the agent can make a maximum of 4 steps in each episode, then the number of neural network inputs sets to 4, and at the beginning, the values of all inputs are 0.

An example is given in Fig. 5. In this example, the agent is allowed to move 4 steps. Hence the number of the neural network's inputs will be 4. In the following, assume that the agent will choose the rules 2,1,2 and 1, respectively (Fig. 5 (a)). Thus, the first input of the neural network at the first step will be 2 (Fig. 5 (b)). At the second step, the second input's value changes to 1 (Fig. 5 (c)). The third input will change to 2 (Fig. 5 (d)). In the last step of the agent, the fourth input of the neural network will change to 1, and the agent cannot take any move anymore. Thus, in the end, the inputs of the neural network will be 2,1,2 and 1, respectively (Fig. 5 (e)), just the same as the order of selection rules in (Fig. 5 (a)).

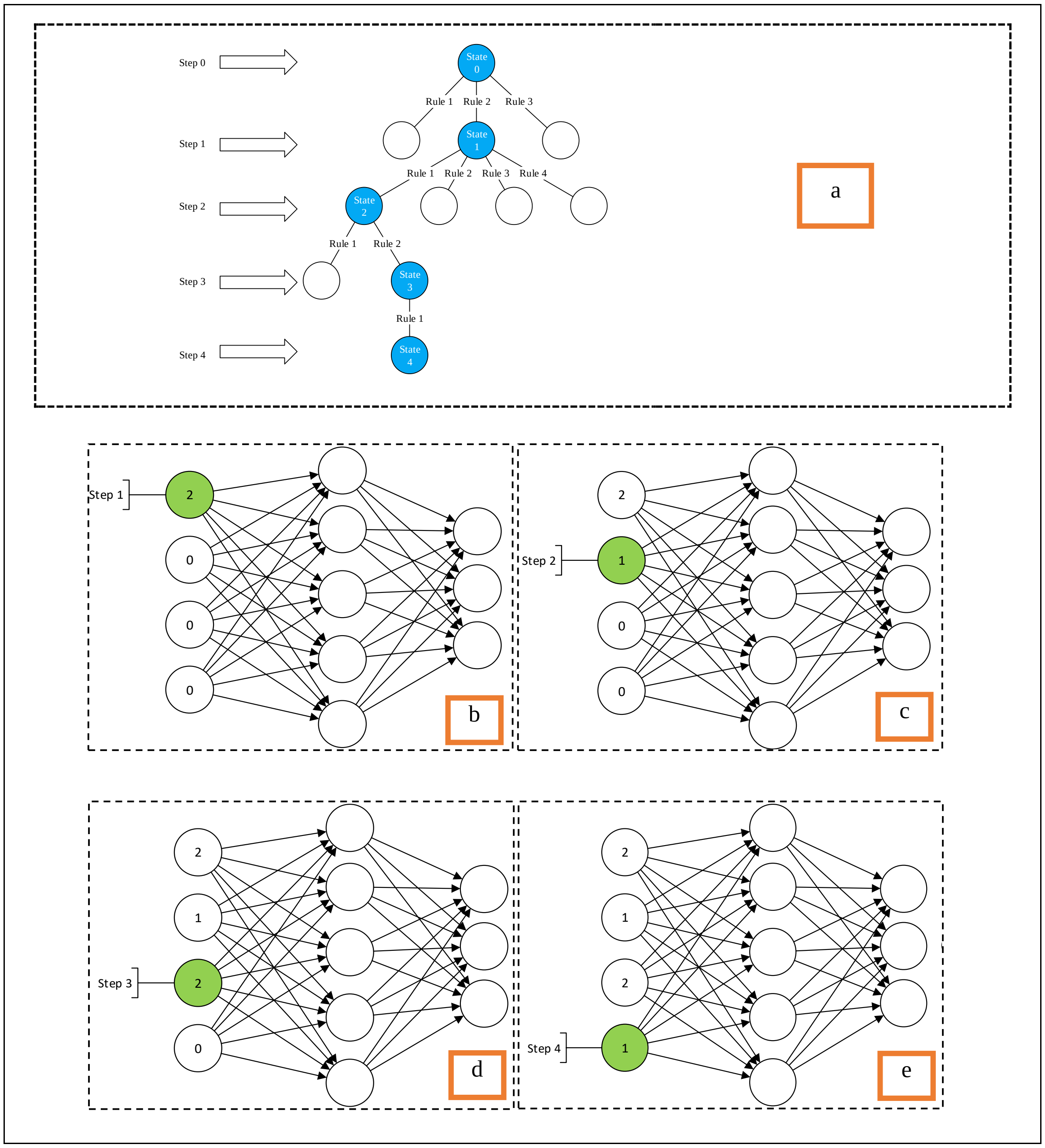


Fig. 5. Example of how to select features.

4.4. Handling Invalid Actions

The number of applicable rules in each state may vary. As a result, the output of a neural network(i.e., the list of Q-values for all rules) may contain some rules that cannot be applied in some states. Also, according to the epsilon-greedy policy, random actions should be selected from the applicable rules in that state. For example, in the state 0 in Fig. 5 (a), only 3 rules can be applied, but number 5 is randomly selected. Due to the absence of rule 5, the agent cannot take any step. Also, due to the fixed structure of the neural network, the number of outputs cannot be dynamic and cannot be changed at any time. Two common ways to solve this problem:

- Give negative rewards for selecting any invalid rule and not change the current status after invalid selection. The disadvantage of this method is the possibility of storing many negative and repetitive experiences in experience replay memory and therefore has a very negative effect on learning the neural network.
- Limit random numbers which are generated at the beginning of the system to gather experiences. Afterwards, create a neural network with a large number of output neurons, filter the output number of the neural network and select the highest value of Q among the valid actions.

4.5. Double Deep Q-Network Implementation

The structure of the proposed approach is presented in Fig. 6, and the overall DDQN implementation algorithm in the GROOVE toolset is given in Algorithm 2. In line 1, the goal state, the number of algorithm episodes and the number of the agent's steps in each episode are received from the user. In line 2, it checks if the reachability property is satisfied, then the path from the initial state leading to the goal state will be returned. In line 3, experience replay memory $\mathcal{D}$ with capacity $\mathcal{N}$ and in line 4, the main neural network Q_θ and the target neural network $Q_{\theta'}$ with weight 0, are initialized. In line 5 to 7, for each episode, the environment is restarted at first. After a restart, the current state is set to the first state of the environment. In lines 8 to 10, it checks that if the initial state is the same as the goal state and property is satisfied, then the path leading to that state is shown to the user as the output.

In lines 12 to 15, at first, the agent will get a list of applicable rules on the current state and select a random action with the probability of ε. Otherwise, If the probability is equal to $1 - \varepsilon$, the next best operation is extracted using the main neural network. In lines 15 to 19, the agent at first performs the selected action, and then the environment rewards the agent and pushes the agent to the next state. It is now checked that if the next state is the same as the goal state, the path leading to that state as an output will be returned. Otherwise, the agent stores its last step as an experience in the experience replay memory $\mathcal{D}$. In lines 20 to 24, a batch of experience is extracted from experience replay memory $\mathcal{D}$ to train the neural network. According to lines 22 and 23, if the next state is equal to the terminal state, the target value will equal the amount of reward. Otherwise, the best action of the next state is selected by the main neural network, and then its Q value is estimated using the target neural network. Finally, it is added with the reward value. In line 25, to synchronize what the agent has learned from the environment, the weight of the target neural network will be transferred to the main neural network. Finally, in line 26, the current state is set to the next state, and if it is possible, the value of ε is decreased.

Algorithm 2 The DDQN based approach

```
1:  Input: $g$ = Goal, M = Episodes, N = Steps
2:  Output: $result$ = if it finds the path, returns it otherwise returns null
3:  Initialize experience replay memory $\mathcal{D}$ to capacity $\mathcal{N}$
4:  Initialize primary network $Q_\theta$ and target network $Q_{\theta'}$ with weights 0
5:  for episodes = 1 to M do
6:      Restart environment
7:      $s_t$ = initial state graph
8:      if $s_t$ = $g$ then
9:          Return a path from the initial state leading to the goal
10:     end if
11:     for steps = 1 to N do
12:         matches = get the list of available rules for the current state
13:         with probability $\varepsilon$ select a random action $a_t$ max to the count of matches
14:         otherwise select $a_t = argmax\ Q_\theta(s_t)$ to count of matches
15:         Execute $a_t$, observe the next state $s_{t+1}$ and set reward $r_t = R(s_t, a_t)$
16:         if $s_t$ = $g$ then
17:             Return a path from the initial state leading to the goal
18:         end if
19:         Store $(s_t, a_t, r_t, s_{t+1})$ in experience replay memory $\mathcal{D}$
20:         batch = Sample random batch from $\mathcal{D}$
21:         foreach $(s_j, a_j, r_j, s_{j+1})$ in batch
22:             if $s_{t+1}$ = Terminal State then Target = $r_j$
23:             Otherwise Target = $r_j + \gamma Q_{\theta'}(s_{j+1}, argmax_a Q_\theta(s_{j+1}, a))$
24:         end for
25:         Reset $Q_{\theta'}$ weights = $Q_\theta$
26:         set $s_t = s_{t+1}$ and if $\varepsilon > \mathbf{threshold}$ then reduce $\varepsilon$
27:     end for
28: end for
```

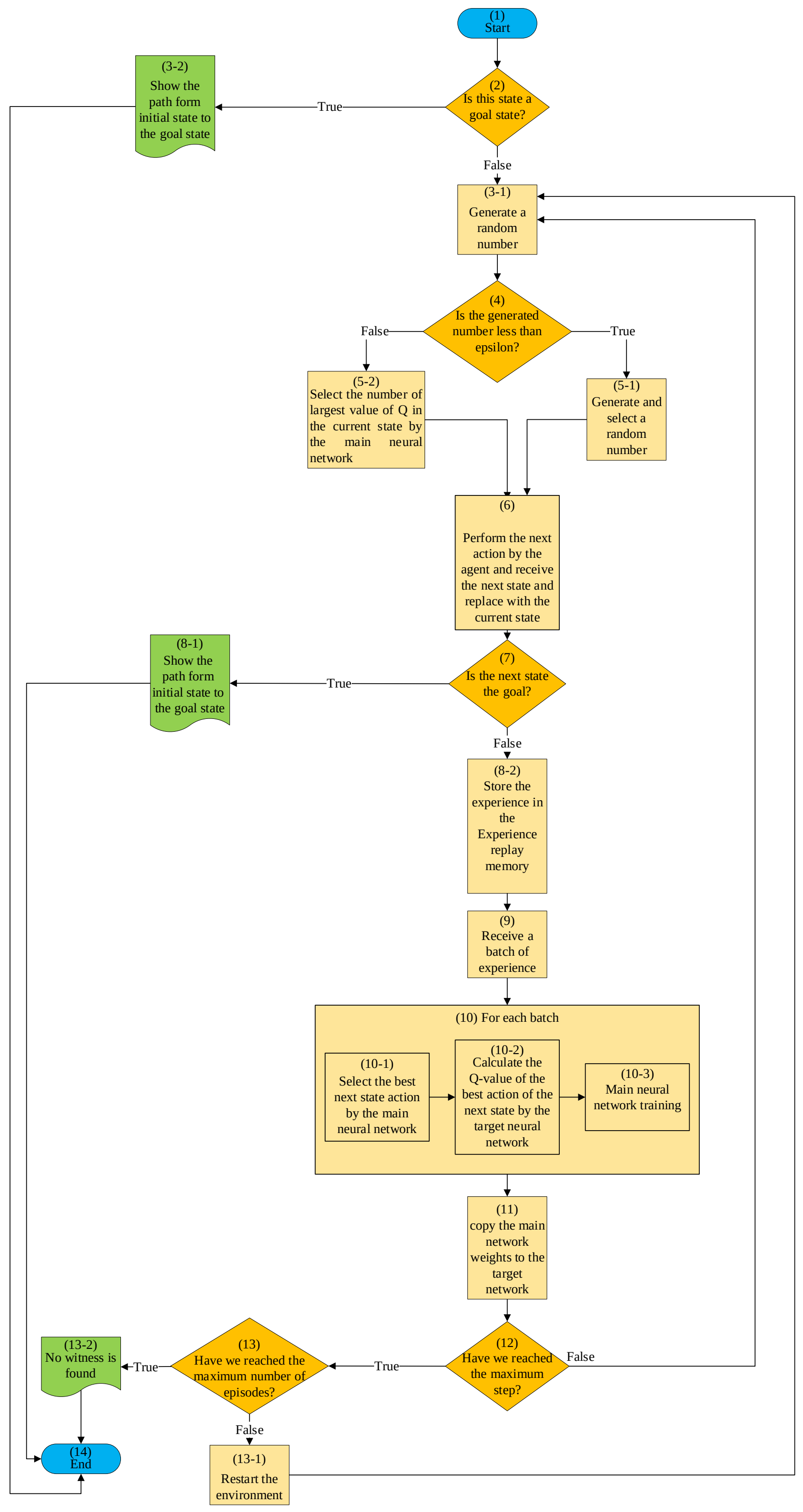

(1) Start
(2) Is this state a goal state?
True
(3-2) Show the path form initial state to the goal state
False
(3-1) Generate a random number
(4) Is the generated number less than epsilon?
False
True
(5-2) Select the number of largest value of Q in the current state by the main neural network
(5-1) Generate and select a random number
(6) Perform the next action by the agent and receive the next state and replace with the current state
(7) Is the next state the goal?
True
(8-1) Show the path form initial state to the goal state
False
(8-2) Store the experience in the Experience replay memory
(9) Receive a batch of experience
(10) For each batch
(10-1) Select the best next state action by the main neural network
(10-2) Calculate the Q-value of the best action of the next state by the target neural network
(10-3) Main neural network training
(11) copy the main network weights to the target network
(12) Have we reached the maximum step?
False
True
(13) Have we reached the maximum number of episodes?
True
(13-2) No witness is found
False
(13-1) Restart the environment
(14) End

Fig. 6. The structure of the proposed approach.

4.6. Complexity

As proposed in Algorithm 2 The DDQN based approach, the proposed approach divides into three main components. The reinforcement learning body, neural network and reward function.

- **Computational Complexity**

The body of reinforcement learning itself contains two loops. So the computational complexity will be $O(m * n)$ where m is the number of episodes and n is the number of steps.

The neural network is the main component of the proposed approach. Since we use linear activation function (e.g. ReLU) in neural networks, the output is a composition of several linear mappings of the input vector. For each batch training, we use two same neural networks. The feedforward propagation algorithm in neural networks is as follows. Assume we have t training examples. Propagating between layers are matrix multiplication. So it has $O(i * h * t)$ computational complexity where *i* and *h* is the number of input and hidden layer neurons accordingly. Then we apply the activation function, and this has $O(h * t)$ computational complexity because it is a linear activation function (e.g. ReLU) that apply to hidden layer neurons. Till now, we have $O((i * h * t) + (h * t)) = O(h * t * (t + 1)) = O(h * i * t)$. After that, we go from the hidden layer to the output layer with $O(h * o * t)$, where o is the number of output neurons. In total, we have $O(i * h * t + h * o * t) = O(t * (ih + ho))$.

The general reward function is explained in 4.2. Reward Engineering. It is $O((a * b) + c)$ which a is current state elements, *b* is goal state elements, and *c* is all pairs in state space.

So in total, we have $O(m * n) + O\big(t * (ih + ho)\big) + O((a * b) + c)$.

- **Storage Complexity**

As mentioned in 3.4. Deep Reinforcement Learning Using Double Deep Q-Network, one of the benefits of deep reinforcement learning is eliminating the overhead of the table of knowledge.

The amount of memory that is allocated to a neuron is $O(i)$, where i is the number of inputs. The reason is that a neuron has one weight per input. The hidden layer has h neurons. If the number of inputs neurons will be n, it needs $h * O(i)$ units of memory. Also, The output layer has *o* neurons with h inputs each, hence $o * O(h)$. So in total, we have $h * O(n) + o * O(h) = O(h * n) + O(o * h) = O((n + o) * h)$

We need to store all intersection pairs of current and goal states in the general reward function. So we have a nested loop to check for all elements, but we need one variable such that its maximum size will eventually be equal to the minimum element count of two states. So we have $O(\min(a, b))$ which a is current state elements, *b* is the goal state.

So in total, we have $O\big((n + o) * h\big) + O(\min(a, b))$.

As mentioned above, the computational and storage complexity can be reduced or increased and depend on neural network activation function, input, hidden and output layer neurons, and reward function.

5. Experimental Results

This section presents the experimental results of DDQN implementation in the GROOVE open-source toolset. This evaluation was performed on a system with 16GB of DDR3 main memory and an Intel Core i7 processor model 4700 MQ with a frequency of 2.40 GHz. At first, to find the optimal number of hidden layers in the neural network, the proposed approach is evaluated using two case studies, and in order to increase the accuracy of the results, each experiment was performed 3 times. In the next step, to determine the optimal number of hidden layer neurons, the previous test is performed again, this time with the obtained number of the hidden layer from the previous test. Finally, after determining the final structure of the neural network, the proposed solution was tested on 5 case studies, which again, to achieve more accurate results, each experiment was performed 10 times separately. In addition, the maximum running time for each experiment is 15 minutes. In order to better compare and understand the performance of the proposed approach, these experiments are performed on the genetic algorithm (GA) [3], nBOA [5], particle swarm optimization (PSO) [30] and IDA* under the same conditions.

The results of the experiments are evaluated in terms of (1) the average running time to reach the goal, (2) the shortest path to reach the goal, (3) the least explored states to reach the goal and (4) the number of witnesses in all of the runs for a problem. The terms of DDQN and DDQN* in the tables of this section is referred to the proposed approach with general and dedicated rewards, respectively, which will be explained separately for each problem. Also, as previously explained, the general reward function is the same as the similarity function that is used in other approaches, such as GA and nBOA [12].

5.1. Case Studies

First, we should review the used case studies in the evaluation.

- **8-Puzzle Problem**: In this problem, there is a 3×3 puzzle in which 8 of 9 cells consist of tiles with random numbers from 1 to 8, and the other cell is empty. The empty cell can be moved between adjacent cells. The goal is to find an arrangement in which the numbers of tiles have an ascending order. In DDQN* mode, we used Manhattan distance plus the current agent's depth as the reward. The first arrangements, the second arrangements, the third arrangements and the fourth arrangements in the evaluations of this case study are shown in Fig. 7(a), Fig. 7(b), Fig. 7(c), Fig. 7(d), respectively.

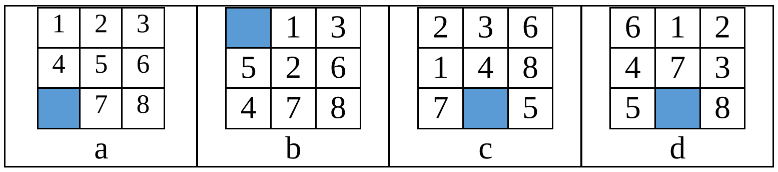


Fig. 7. Different arrangements of 8-Puzzle problem.

- **N-Queen problem:** There is an N×N chessboard and N queens in this problem. The goal is to find an arrangement in which no queen can guard another one. Two queens can guard each other when their columns, rows or diameters are the same. Thus, an arrangement is acceptable in which all queens have different columns, rows and diameters.

- **Blocks World problem:** As mentioned in section 4.2. Reward Engineering, in this problem, there are three various colours of blocks, and the goal is to place the same colour blocks on top of each other. There are some rules in this problem.
    - Only one block may be moved in each step.
    - Any block that is placed under another block cannot be moved.
    - Different colour blocks cannot be moved on top of each other.
- **Dining philosophers problem:** As mentioned earlier in section 3.2. Graph Transformation System, this problem is about some philosophers that there is a fork between each pair of them. Each philosopher thinks, gets hungry, takes left and right forks, respectively, and releases both forks after eating. This process will be repeated indefinitely. The ultimate goal is a terminal state, in which all philosophers have held their left fork, and therefore, no philosopher can hold his right fork anymore to start eating. As a result, the system is in the terminal state (or deadlock in this case), which means that no more rules can be applied to the graph.
- **Snake problem**: This old problem was introduced in the world planning competitions. These competitions are held annually by the International Conference on Automated Planning and Scheduling, and this case study was one of the problems in the 2018 competition. In this problem, a snake moves on the ground and eats apples that exist in some location (i.e., cell) of this ground. By eating each apple, the snake's length will increase by 1, and a new apple will grow in a different cell. In order to eliminate uncertainty in this problem, the apples grow in a predetermined location. Ultimately, the goal state is the state in which the snake has eaten all the apples.

5.2. Required Parameters

The parameters used in the experiments are presented in Table 1.

- *Episodes* and *Steps* parameters depend on the problem. A higher value of episodes results in a more accurate and time-consuming result. Because the proposed approach will be repeated and on each start, the knowledge will increase. So the result will converge. The Steps parameter is the same as depth in graphs. In some problems, the goal cannot be found in lower depth, and due to this, the agent should go further and take more steps.
- *Experience Replay Memory Size* is set to 1000. A higher value of Experience Replay Memory Size cause more storage usage and also, a lower value of this parameter causes the agent to forget about past experiences.
- The *Discount Factor* allows modelling the agent behaviour with respect to the future rewards. In particular, a high discount factor (like 0.1) will make the agent greedy such that it will choose actions that provide the highest immediate reward. It is because future rewards will be heavily discounted.

Hence they will have a limited impact when the cumulative reward is computed. So The value is set to 0.95.

- *The Minimum Value Of Epsilon* can be between 0 and 1. For a more greedy approach, this value must be near 0.
- As the epsilon first is set to 1, the *Reduction Rate Of Epsilon In Every Step* value is set to 0.995. A higher value of this parameter causes the agent to act more randomly at first and slowly become greedy.
- As said in section 4.1. Neural Network, *Learning Rate* controls how quickly or slowly a neural network model learns a problem. In the proposed approach, the most traditional value is set.
- The *Batch Size* parameter shows the size of the training set in each train of neural network. If this parameter sets too high, the neural network's prediction may be more accurate, but it takes more storage and time. With using a small batch size, future predictions will be poor. So there must be a balance between speed and accuracy.
- *Max Action Output* value is mentioned in section 4.1. Neural Network. A higher value of this parameter increases memory consumption and wastes time. Setting Max action output to a lower value limits the actions in state space (e.g. applied rules).

Table 1: The initial parameters for the proposed approach.

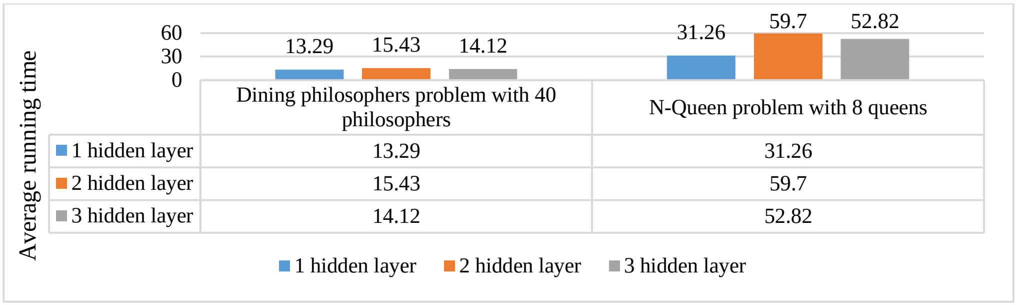
Name	Description	Value
Episodes	The number of iterations of the algorithm	300
Steps	The number of steps in each iteration (depending on the problem). Also, it is the number of neural network's inputs	100, 150, 200, 500, 700 or 1000
Experience Replay Memory Size	Experimental replay memory size	1000
Discount Factor	The importance of future rewards	0.95
The Minimum Value Of Epsilon	Minimum Epsilon value for random selection of actions	0.2
Reduction Rate Of Epsilon In Every Step	Reduction value of epsilon per step	0.995
Learning Rate	The neural network learning rate	0.001
Batch Size	Batch size of the training dataset	8
Max Action Output	Maximum number of neural network's outputs	400

5.3. Number of hidden layers and optimal neurons for the neural network

The number of hidden layers and their neurons is a question for which there is no definitive answer, and their optimal number cannot certainly be determined before the experiment. We tested our proposed approach under the same conditions on two different case studies to obtain the optimal number.

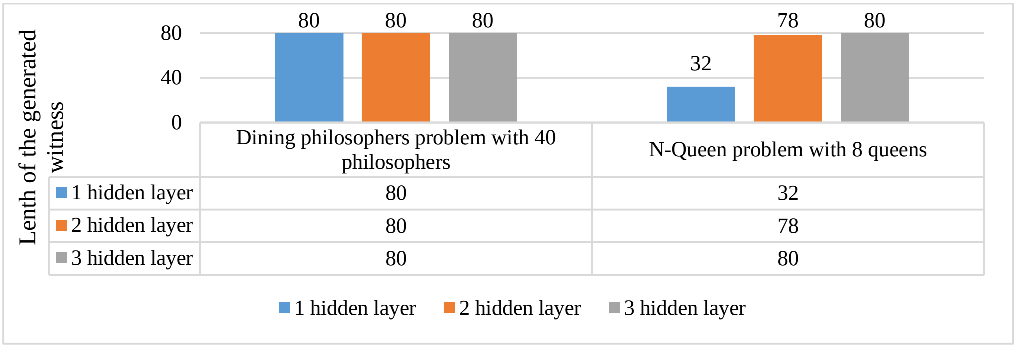
- **The optimal number of hidden layers**

Fig. 8 shows the average running time in the neural network with different layers. As it is shown, the neural network with 1 hidden layer was able to respond to both problems faster than the others. Fig. 9 shows the minimum length of the generated witnesses obtained from the neural network with several different numbers of hidden layers. The neural network with 1 hidden layer is the winner because it generates shorter witnesses than the others. As a result, this paper uses a neural network with 1 hidden layer to evaluate other experiments.



	Dining philosophers problem with 40 philosophers	N-Queen problem with 8 queens
■ 1 hidden layer	13.29	31.26
■ 2 hidden layer	15.43	59.7
■ 3 hidden layer	14.12	52.82

Fig. 8. Comparison of the average running time in the neural network with different numbers of hidden layers.



	Dining philosophers problem with 40 philosophers	N-Queen problem with 8 queens
■ 1 hidden layer	80	32
■ 2 hidden layer	80	78
■ 3 hidden layer	80	80

Fig. 9. Comparison of the length of the generated witness in the neural network with different numbers of hidden layers.

- **The optimal number of hidden layer neurons**

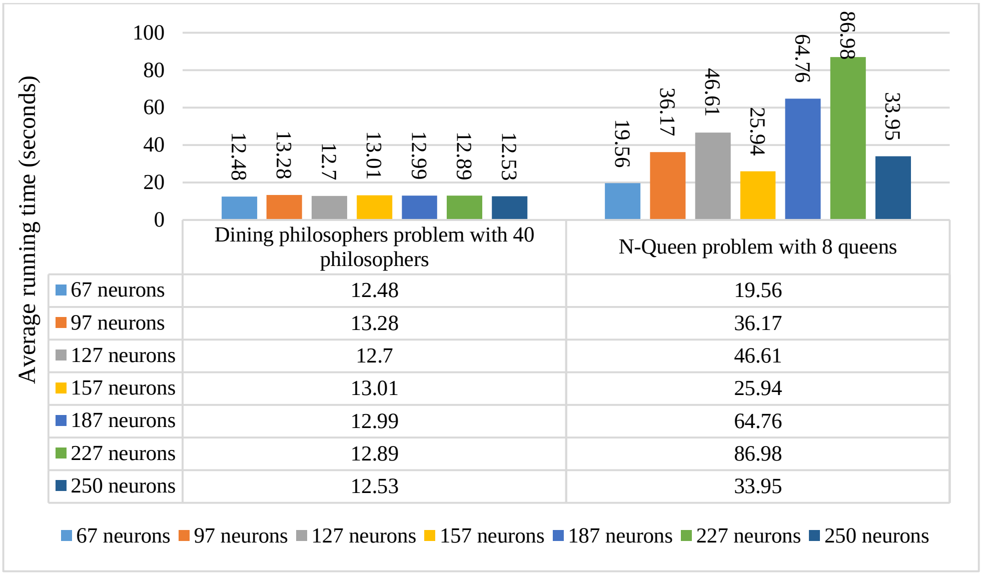
There is no specific rule for choosing the optimal number of neurons in the hidden layer, but some formulas have been proposed to achieve the desired result [31]. In these formulas that are shown in Table 2, we need to define the number of input layer neurons as $\boldsymbol{N_i}$ and in some of them, the number of output layer neurons as $\boldsymbol{N_o}$. As mentioned in section 4.1. Neural Network, 100 and 400 will be used for N_i and N_o respectivly.

Eventually, as it is shown in Table 2, the the optimal number of neurons is between 67 and 250, which is tested separately by changing the number of hidden layer neurons with 30 steps.

Table 2 Result of applying formulas in [31] to get the number of hidden neurons

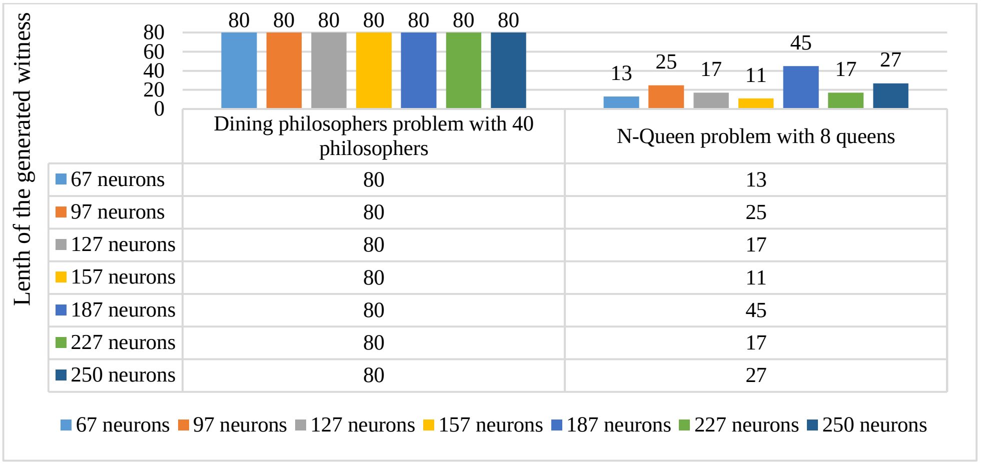
Formula	N_i	N_o	Result
$\leq 2 \times N_i + 1$	100	400	201
$(N_i + N_o)/2$	100	400	250
$2N_i/3$	100	400	67
$\sqrt{N_i \times N_o}$	100	400	200
$2N_i$	100	400	200

In Fig. 10, with 67 neurons, the fastest result was obtained in both case studies. In addition, as shown in Fig. 11, the neural network with 67 neurons in the case of 8 queens was able to take second place. According to the formulas presented in [31], this number of neurons is equal to $\frac{2}{3}$ input neurons and it can balance the speed and optimal length of generated witness; we use this formula to obtain the optimal number of hidden layer neurons in all future experiments.



	Dining philosophers problem with 40 philosophers	N-Queen problem with 8 queens
🟦 67 neurons	12.48	19.56
🟧 97 neurons	13.28	36.17
⬜ 127 neurons	12.7	46.61
🟨 157 neurons	13.01	25.94
🟦 187 neurons	12.99	64.76
🟩 227 neurons	12.89	86.98
🟦 250 neurons	12.53	33.95

Fig. 10. Comparing the average running time in seconds in the neural network with different numbers of neurons.



	Dining philosophers problem with 40 philosophers	N-Queen problem with 8 queens
67 neurons	80	13
97 neurons	80	25
127 neurons	80	17
157 neurons	80	11
187 neurons	80	45
227 neurons	80	17
250 neurons	80	27

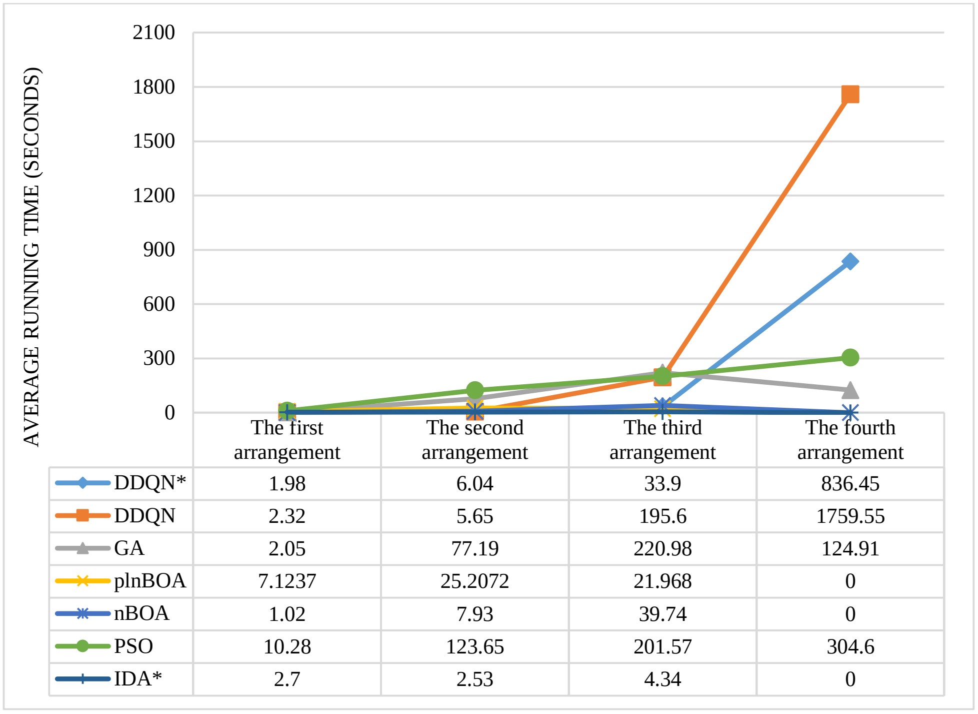
Fig. 11. Comparison of the length of the generated witness in the neural network with different numbers of hidden layers.

5.4. Evaluation

In order to evaluate the proposed approach, five case studies are evaluated. In this section, a brief description of the problem and then the evaluation results are provided.

8-Puzzle problem: Fig. 12 shows the comparison of the average running time in seconds, Fig. 13 shows the comparison of the accuracy (i.e. the number of witnesses), Fig. 14 shows the comparison of the length of the witness, and Fig. 15 shows the comparison of the number of explored states to satisfy the property in the 8-Puzzle problem.

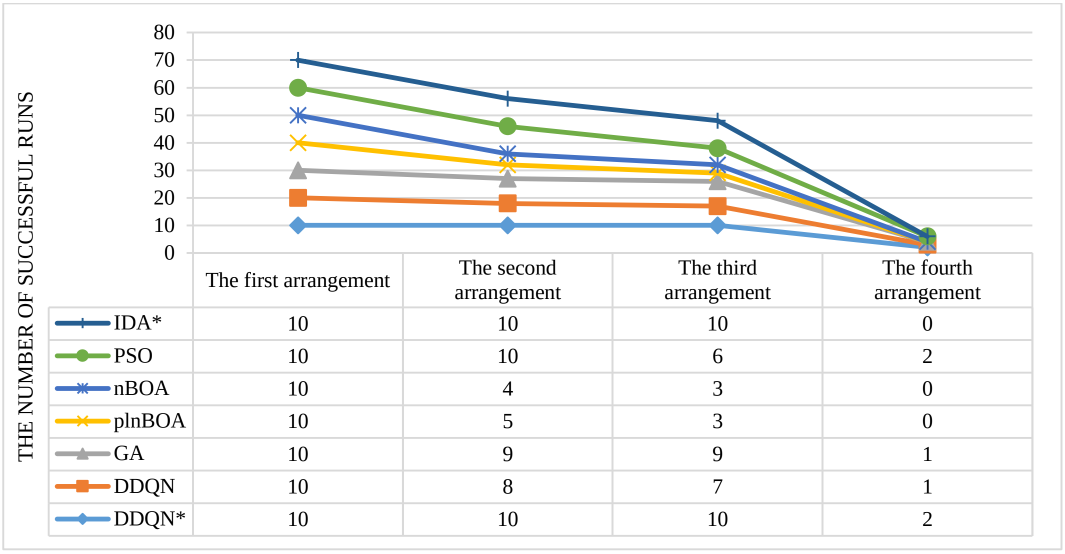
In the case of the N-Queen problem, considering Fig. 12, it is quite clear that DDQN has spent much time to reach the goal state due to the heavy calculation of the reward function in each explored state, while DDQN* is much faster because of its dedicated reward function. The IDA* method performed much better than other approaches in terms of speed in the case of second and third arrangements. However, as the problem becomes complex, the number of explored states increases. Due to the large state space, IDA* may encounter the same amount of rewards along the way, and therefore, more states are needed to be considered because of its exploration method. The nBOA approach works better than DDQN in terms of speed and time because it depends on the rules, and also it does not need to calculate the reward function many times.



	The first arrangement	The second arrangement	The third arrangement	The fourth arrangement
DDQN*	1.98	6.04	33.9	836.45
DDQN	2.32	5.65	195.6	1759.55
GA	2.05	77.19	220.98	124.91
plnBOA	7.1237	25.2072	21.968	0
nBOA	1.02	7.93	39.74	0
PSO	10.28	123.65	201.57	304.6
IDA*	2.7	2.53	4.34	0

Fig. 12. Comparing the average running time in seconds in the 8-Puzzle problem.

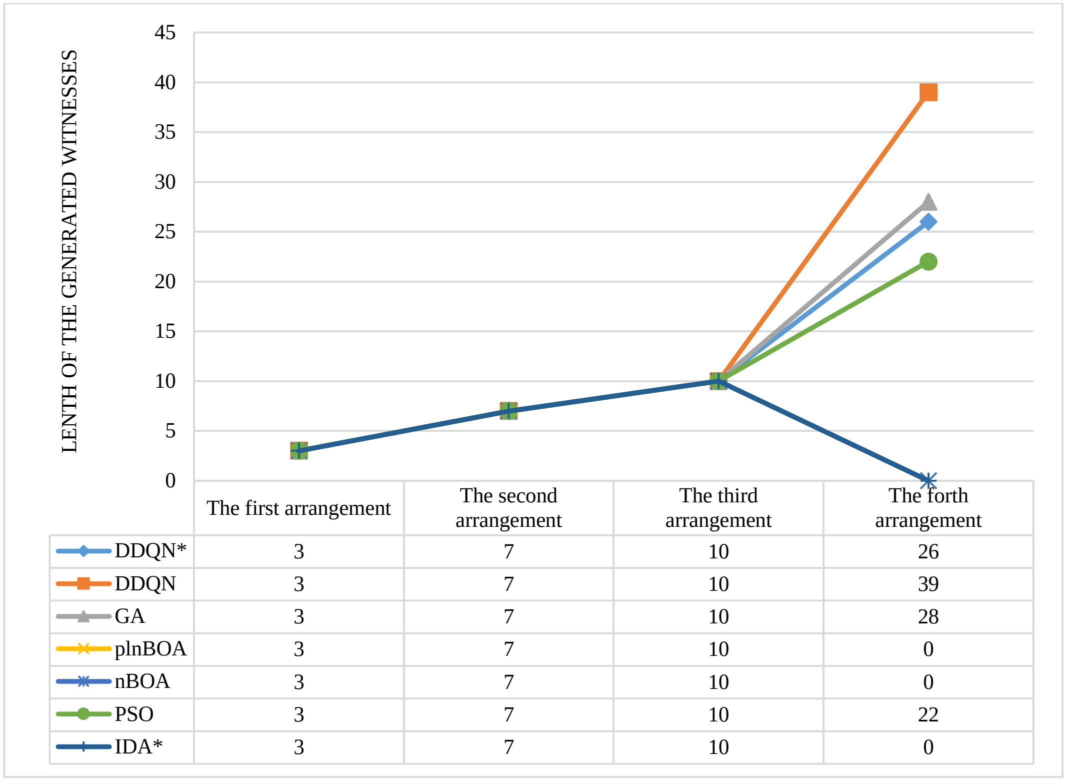
According to Fig. 13, in the third arrangement, the IDA* and DDQN* approaches have succeeded in finding the goal state in all 10 runs. DDQN is also weaker than GA due to its inaccurate reward function. Naturally, the GA has a better chance of reaching the goal state and escaping from the local optimum trap due to its capabilities such as mutation and crossover. However, DDQN performed better than nBOA and PSO. The overall performance of DDQN* because of the impact of its reward function is quite clear.



	The first arrangement	The second arrangement	The third arrangement	The fourth arrangement
IDA*	10	10	10	0
PSO	10	10	6	2
nBOA	10	4	3	0
plnBOA	10	5	3	0
GA	10	9	9	1
DDQN	10	8	7	1
DDQN*	10	10	10	2

Fig. 13. Comparing the ratio of the number of successful runs to the total runs in the 8-Puzzle problem.

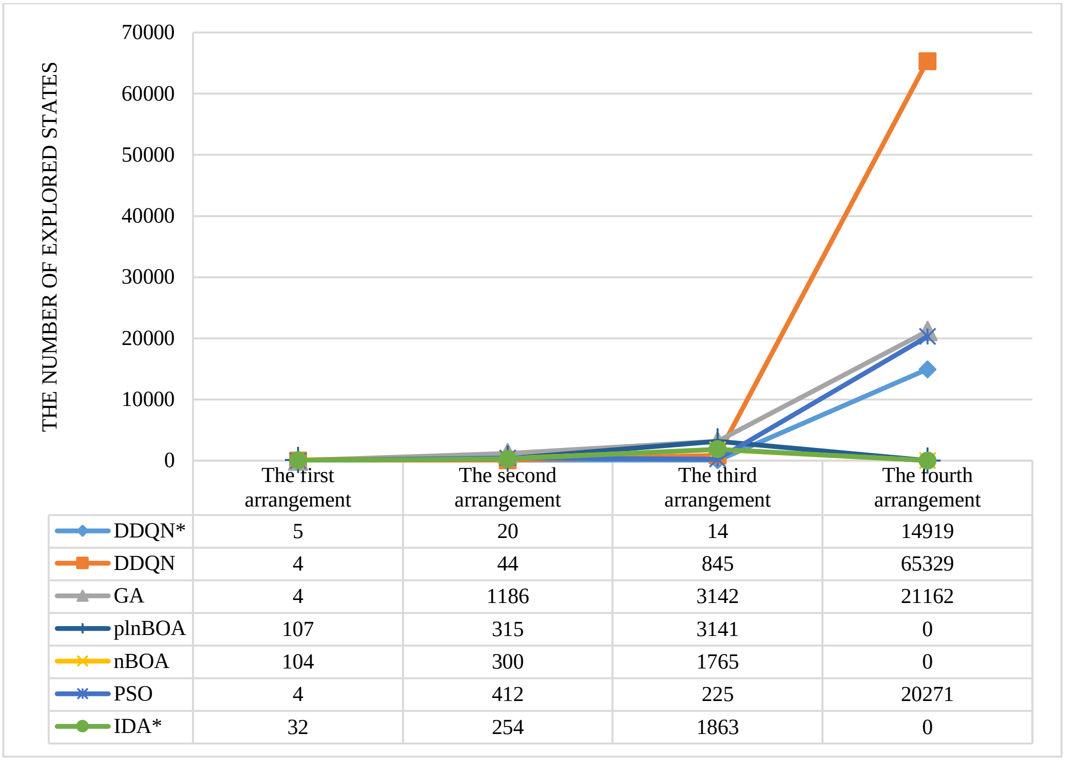
According to Fig. 14, The length of generated witnesses in the first, second and third arrangement is 3, 7 and 10 for all approaches. However, in the fourth arrangement, plnBOA, IDA* and nBOA fail, and the length of the generated witness was 26, 39, 28 and 22 for DDQN*, DDQN, GA and PSO, respectively. According to these results, the PSO approach was able to find the goal at a lower depth due to the particle swarm and more flexibility of this algorithm against the local optimum trap. DDQN* has failed to achieve lower depth targets due to its greedy nature in choosing the best Q value and reducing the random selection rate during the path. Also, nBOA, plnBOA and IDA* fail to find the goal state, which is a big disadvantage for these approaches.



	The first arrangement	The second arrangement	The third arrangement	The forth arrangement
DDQN*	3	7	10	26
DDQN	3	7	10	39
GA	3	7	10	28
plnBOA	3	7	10	0
nBOA	3	7	10	0
PSO	3	7	10	22
IDA*	3	7	10	0

Fig. 14. Comparing the length of the generated witnesses in the 8-Puzzle problem.

As it is shown in Fig. 15, DDQN* and DDQN have been able to find the goal state with the least number of explored states, indicating the performance of these two approaches to find the path to achieve the goal. Especially in more complex arrangements of this problem, it is important to intelligently explore the state space to prevent the state space explosion. However, the number of explored states in nBOA approach is not as optimal as our proposed approach. In a more complex puzzle arrangement, i.e. the fourth arrangement, this time DDQN* and PSO performed better than the other algorithms, and DDQN*, with fewer explored states, was superior to PSO.

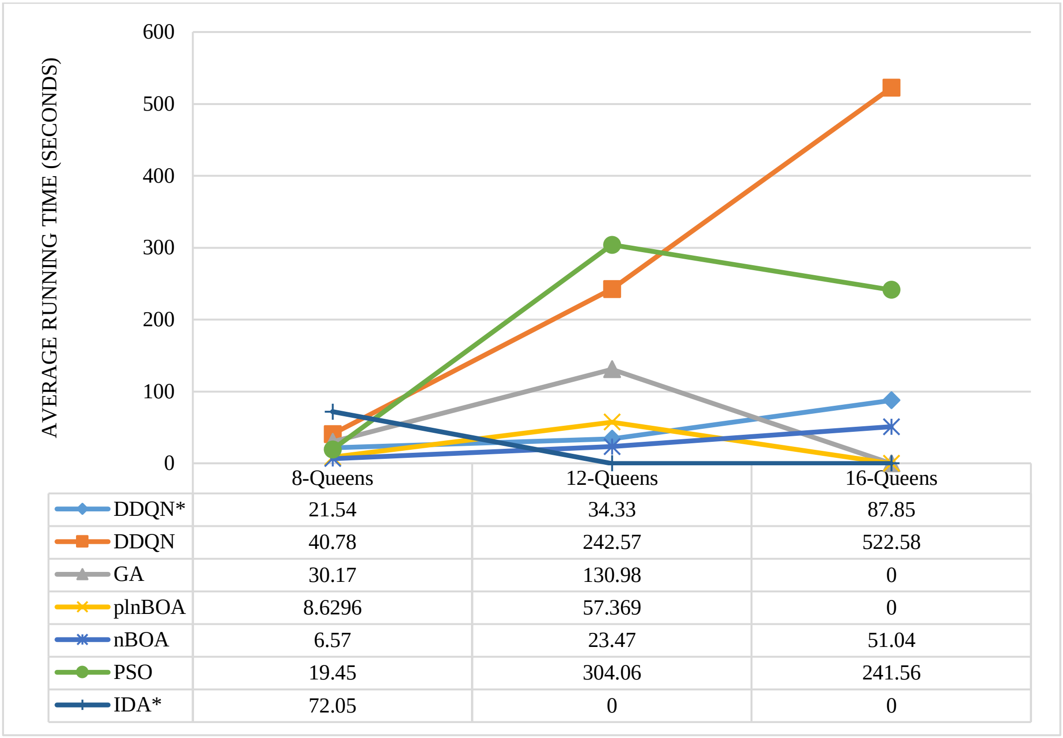


	The first arrangement	The second arrangement	The third arrangement	The fourth arrangement
DDQN*	5	20	14	14919
DDQN	4	44	845	65329
GA	4	1186	3142	21162
plnBOA	107	315	3141	0
nBOA	104	300	1765	0
PSO	4	412	225	20271
IDA*	32	254	1863	0

Fig. 15. Comparing the number of explored states in the 8-Puzzle problem.

N-Queen Problem: Fig. 16 shows the comparison of the average running time in seconds, Fig. 17 shows the comparison of the accuracy (i.e., the number of witnesses), Fig. 18 shows the comparison of the length of the witness and Fig. 19 shows the comparison of the number of explored states to satisfy the property in the N-Queen problem.

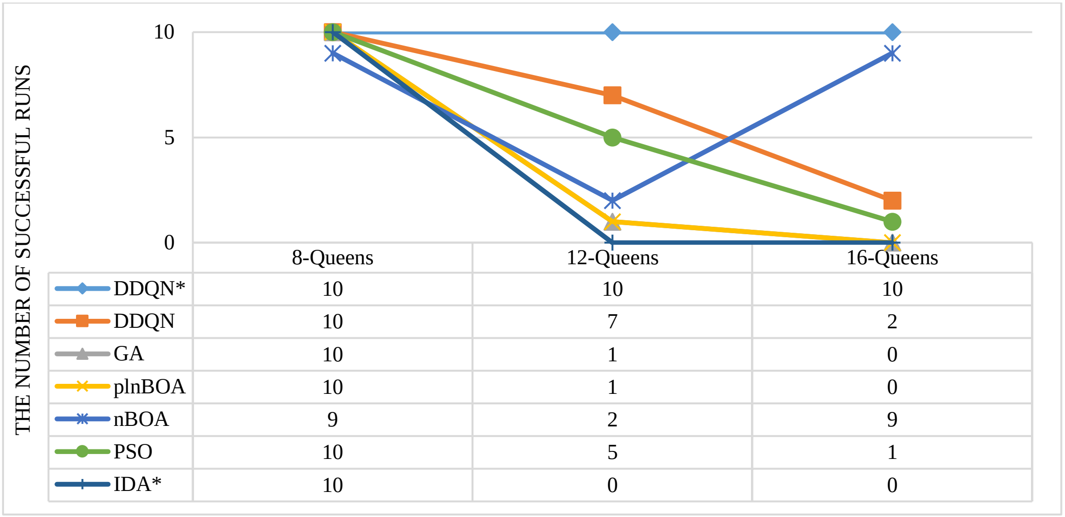
In the case of the N-Queen problem, According to Fig. 16, the fastest approach is the nBOA approach. Because this approach depends on the rules and thus, it does not need to calculate the reward function many times. Also, because of the lower need to use the fitness function to achieve the goal in GA, this approach is faster than DDQN.



	8-Queens	12-Queens	16-Queens
DDQN*	21.54	34.33	87.85
DDQN	40.78	242.57	522.58
GA	30.17	130.98	0
plnBOA	8.6296	57.369	0
nBOA	6.57	23.47	51.04
PSO	19.45	304.06	241.56
IDA*	72.05	0	0

Fig. 16. Comparing the average running time in seconds in the N-Queen problem.

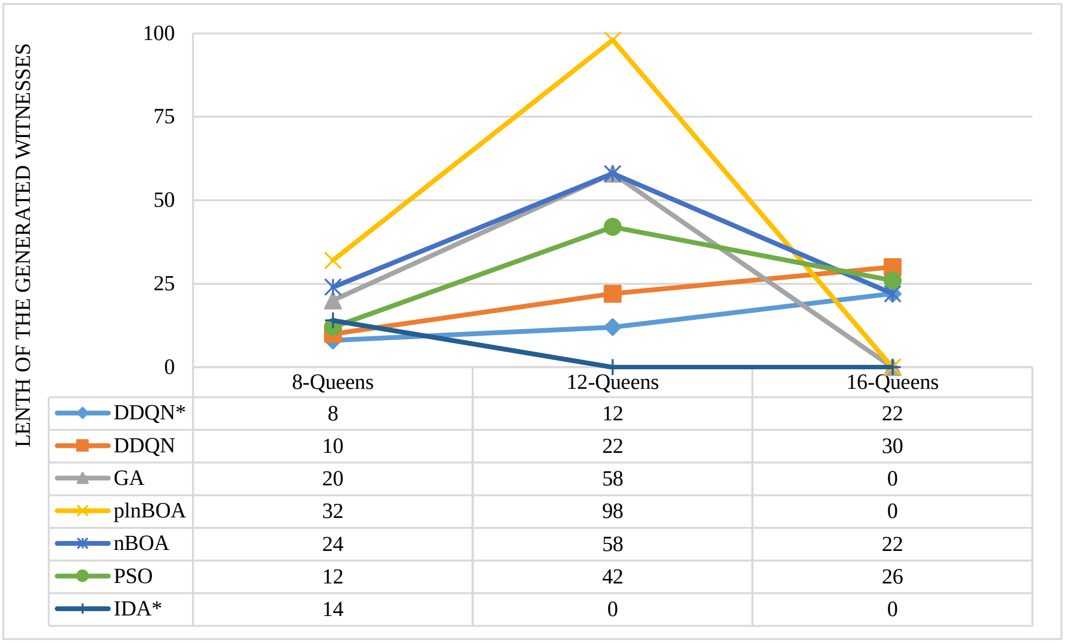
Fig. 17 shows that, In the case of 12 queens, DDQN* can find the goal in all 10 experiments, and DDQN came in second place with 7 answers. However, GA can only find 1 witnesses in large state spaces out of 10. In the case of 16 queens, nBOA shows that there is always the possibility of a better performance in terms of accuracy in the large state spaces than the smaller ones. It also performs much better than plnBOA. plnBOA, GA and IDA* fail to find the goal state in 16 queens.



	8-Queens	12-Queens	16-Queens
DDQN*	10	10	10
DDQN	10	7	2
GA	10	1	0
plnBOA	10	1	0
nBOA	9	2	9
PSO	10	5	1
IDA*	10	0	0

Fig. 17. Comparing the ratio of the number of successful runs to the total runs in the N-Queen problem.

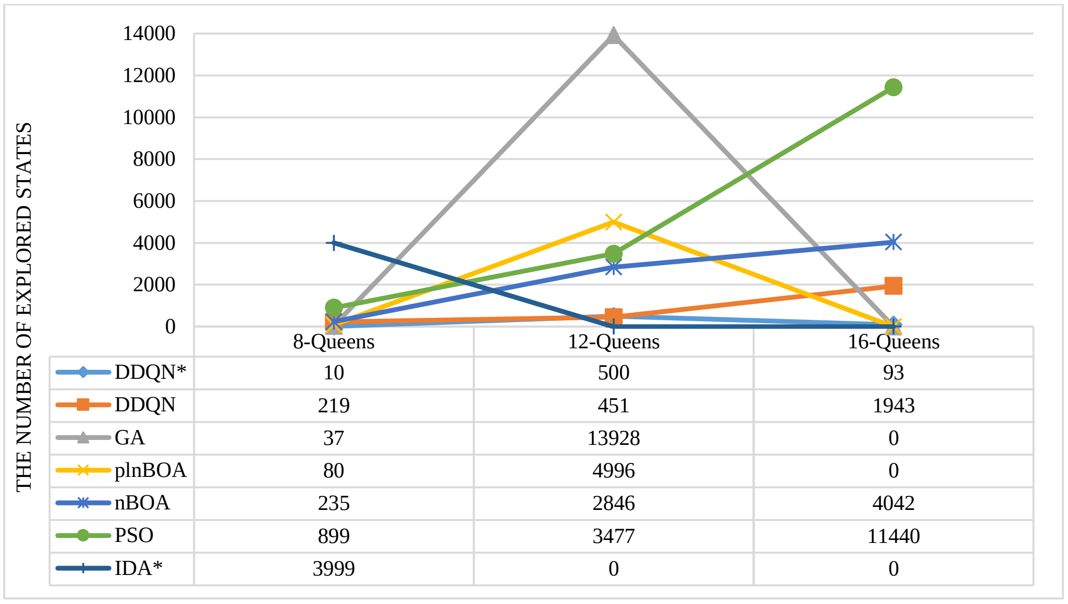
According to Fig. 18, the DDQN* approach achieves the shortest generated witness in all 8, 12 and 16 queens cases. Except for 16-Qunnes, the DDQN approach also performs well in terms of the length of the generated witnesses.



	8-Queens	12-Queens	16-Queens
DDQN*	8	12	22
DDQN	10	22	30
GA	20	58	0
plnBOA	32	98	0
nBOA	24	58	22
PSO	12	42	26
IDA*	14	0	0

Fig. 18. Comparing the length of the generated witnesses in the N-Queen problem.

Also, As shown in Fig. 19, DDQN* and DDQN performed much better than other approaches in terms of fewer explored states. Finally, DDQN* once again shows a better performance in terms of fewer explored states to reach the goal state and more accuracy (Fig. 17).

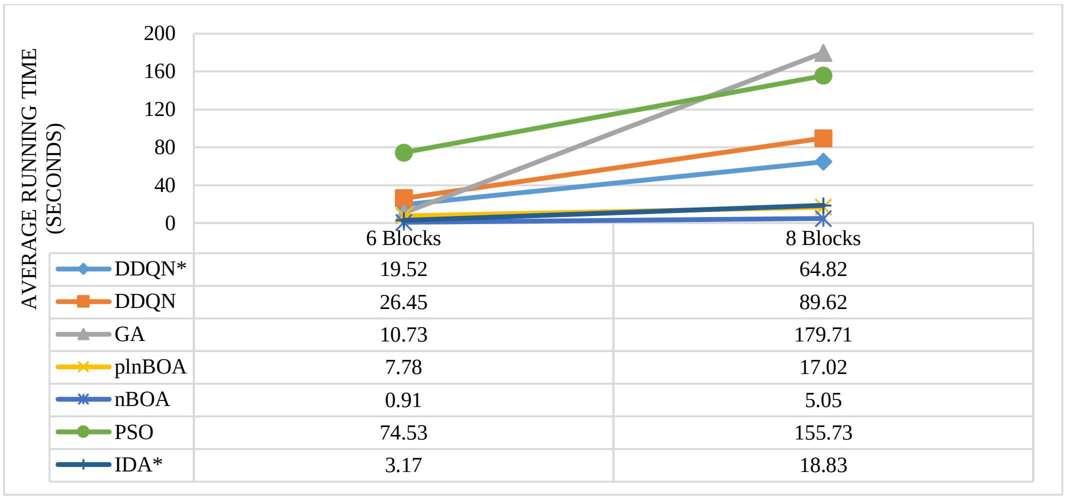


	8-Queens	12-Queens	16-Queens
DDQN*	10	500	93
DDQN	219	451	1943
GA	37	13928	0
plnBOA	80	4996	0
nBOA	235	2846	4042
PSO	899	3477	11440
IDA*	3999	0	0

Fig. 19. Comparing the number of explored states in the N-Queen problem.

Blocks World Problem: Fig. 20 shows the comparison of the average running time in seconds, Fig. 21 shows the comparison of the accuracy (i.e. the number of witnesses), Fig. 22 shows the comparison of the length of the witnesses and Fig. 23 shows the comparison of the number of explored states to satisfy the property in the blocks world problem.

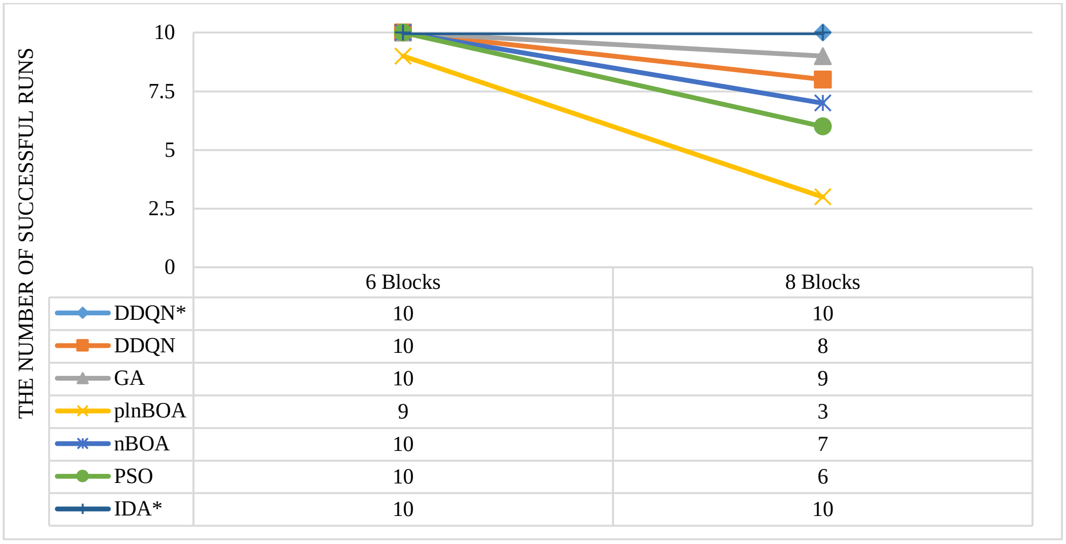
According to the results in Fig. 20, the nBOA approach finds the goal state faster than the proposed approach in the blocks world.



	6 Blocks	8 Blocks
DDQN*	19.52	64.82
DDQN	26.45	89.62
GA	10.73	179.71
plnBOA	7.78	17.02
nBOA	0.91	5.05
PSO	74.53	155.73
IDA*	3.17	18.83

Fig. 20. Comparing the average running time in seconds in the blocks world problem.

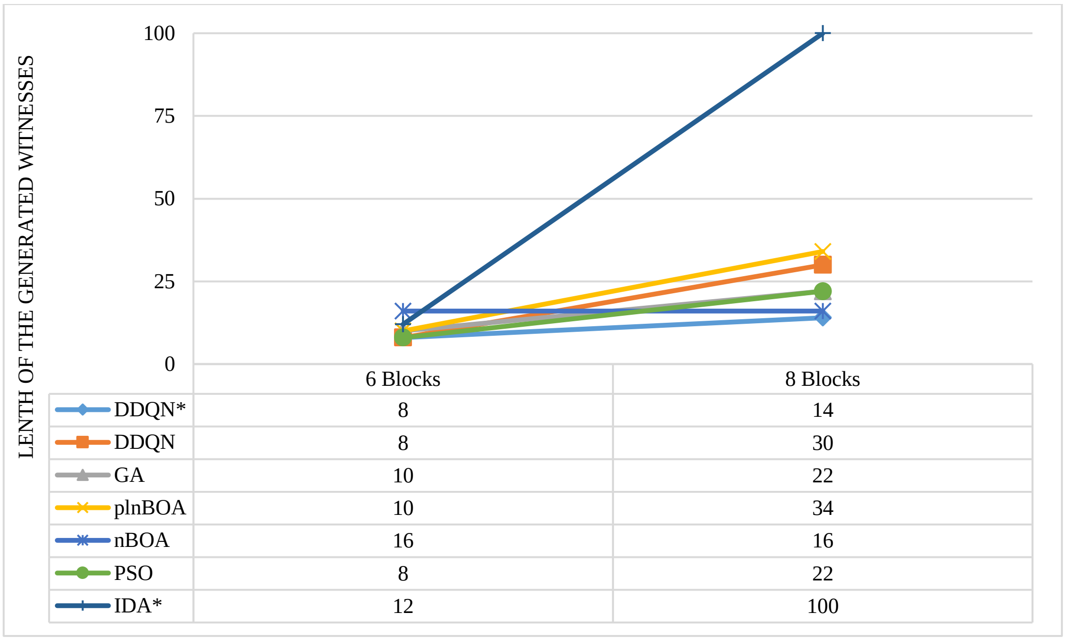
Although DDQN performs worse in terms of speed than nBOA, according to the results in Fig. 21, it can find 8 out of every 10 runs, which this number is 7 in nBOA.



	6 Blocks	8 Blocks
DDQN*	10	10
DDQN	10	8
GA	10	9
plnBOA	9	3
nBOA	10	7
PSO	10	6
IDA*	10	10

Fig. 21. Comparing the ratio of the number of successful runs to the total runs in the blocks world problem.

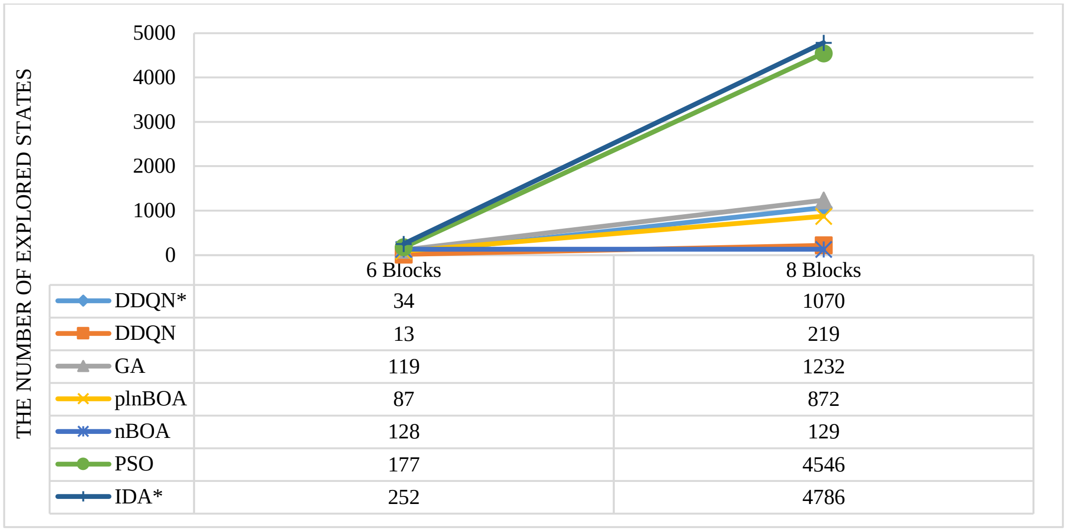
As shown in Fig. 22, the DDQN* approach finds the shortest witness among the other approaches. In addition, the IDA* method will always provide the same witness. It is because of the fact that the reward of a path is constant, and this approach relies on the cost of the paths.



	6 Blocks	8 Blocks
DDQN*	8	14
DDQN	8	30
GA	10	22
plnBOA	10	34
nBOA	16	16
PSO	8	22
IDA*	12	100

Fig. 22. Comparing the length of the generated witnesses in the blocks world problem.

According to the results of Fig. 23, when the number of blocks reached 6, nBOA scored a point by exploring the fewer states than the others. In addition, DDQN was able to find the goal state with fewer explored states than DDQN*. The reason could be the overestimation in DDQN* approach. While DDQN* and IDA* both find the goal state in all 10 runs (Fig. 21), the witness in the IDA* approach is not as optimal as DDQN* in terms of length and the number of explored states. Also, plnBOA performs worse than others.

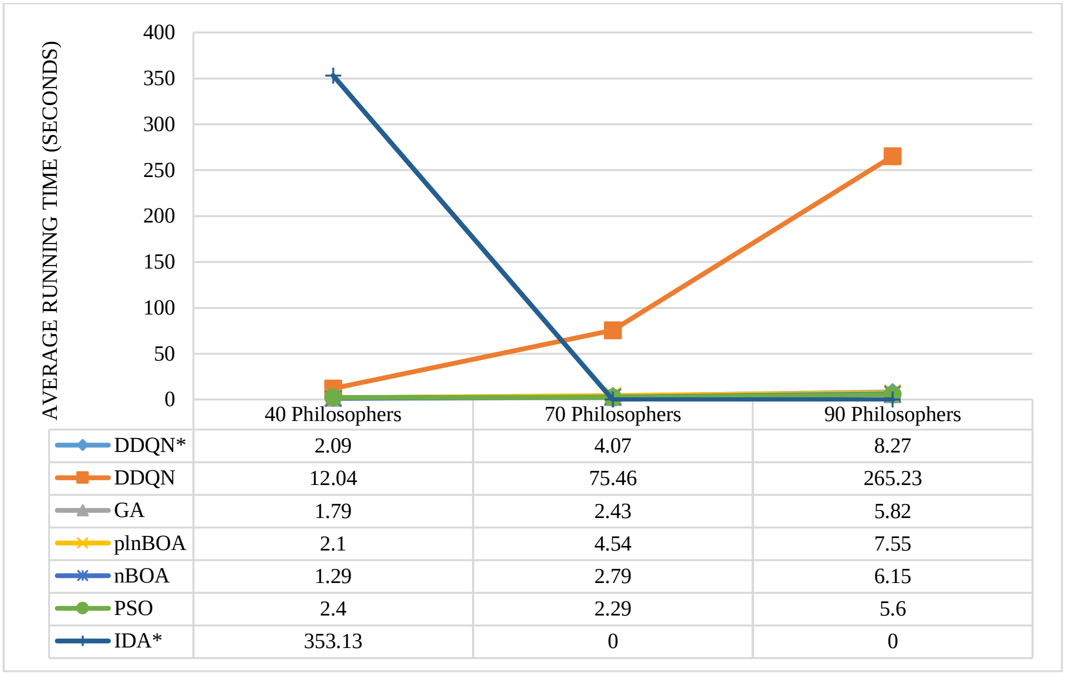


	6 Blocks	8 Blocks
DDQN*	34	1070
DDQN	13	219
GA	119	1232
plnBOA	87	872
nBOA	128	129
PSO	177	4546
IDA*	252	4786

Fig. 23. Comparing the number of explored states in the blocks world problem.

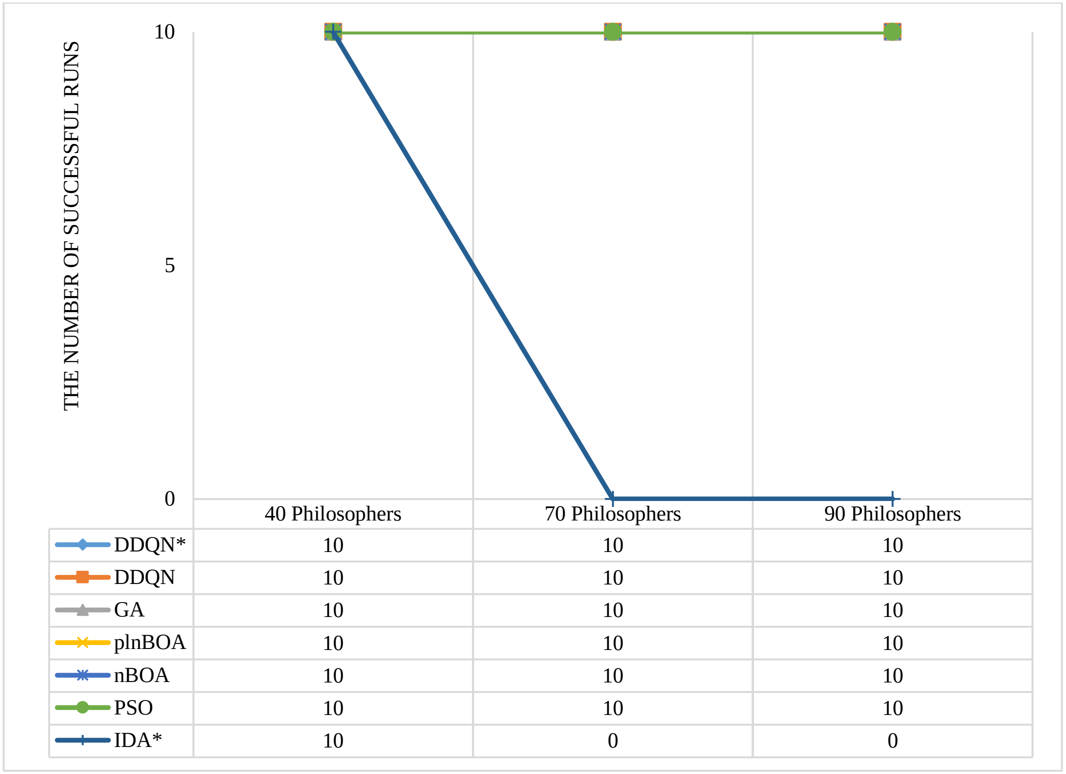
Dining philosophers Problem: Fig. 24 shows the comparison of the average running time in seconds, Fig. 25 shows the comparison of the accuracy (i.e. the number of witnesses), Fig. 29 shows the comparison of the length of the witness, and Fig. 27 shows the comparison of the number of explored states to satisfy the property in the dining philosopher problem.

According to Fig. 24 and Fig. 25, as the state space becomes large, the IDA* fails and cannot find a witness, indicating a drawback of this approach in large state spaces. Thus, IDA* fails in 70 and 90 philosophers problems. However, all six other approaches found the witness in all runs. Also, according to these results, it can be concluded that due to the high usage of reward function in the DDQN approach and also the heavy calculation of this function, this approach is much slower than the others to find the goal state. However, the DDQN approach does not encounter state space explosion.



	40 Philosophers	70 Philosophers	90 Philosophers
DDQN*	2.09	4.07	8.27
DDQN	12.04	75.46	265.23
GA	1.79	2.43	5.82
plnBOA	2.1	4.54	7.55
nBOA	1.29	2.79	6.15
PSO	2.4	2.29	5.6
IDA*	353.13	0	0

Fig. 24. Comparing the average running time in seconds in the dining philosophers problem.

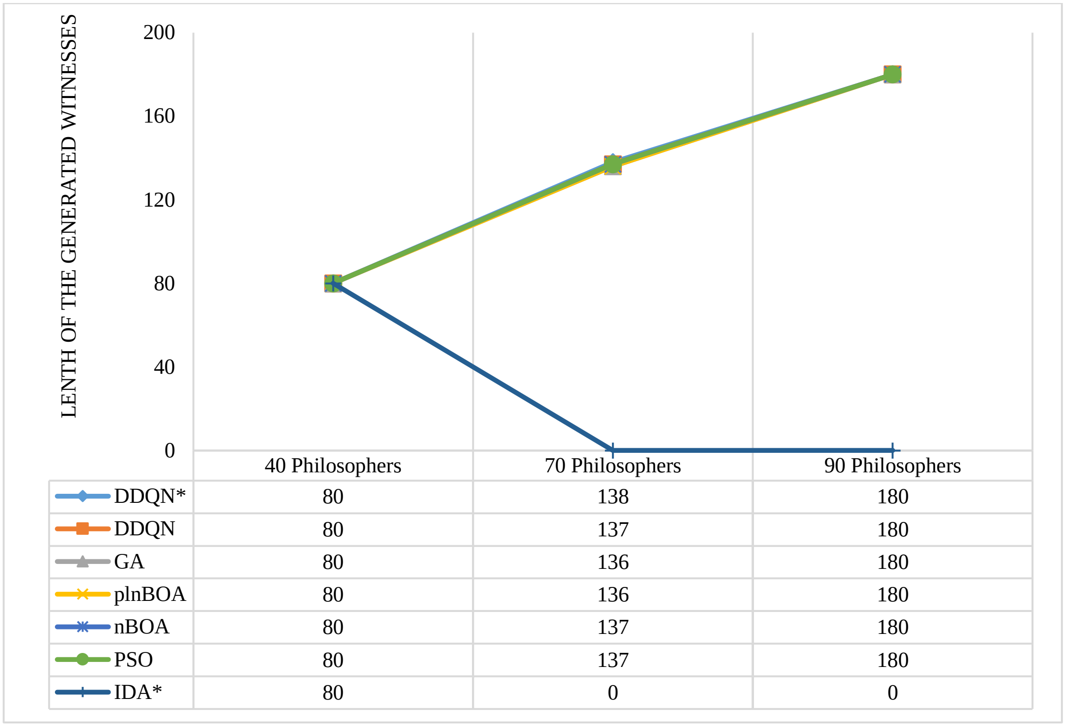


	40 Philosophers	70 Philosophers	90 Philosophers
DDQN*	10	10	10
DDQN	10	10	10
GA	10	10	10
plnBOA	10	10	10
nBOA	10	10	10
PSO	10	10	10
IDA*	10	0	0

Fig. 25. Comparing the ratio of the number of successful runs to the total runs in the dining philosophers problem.

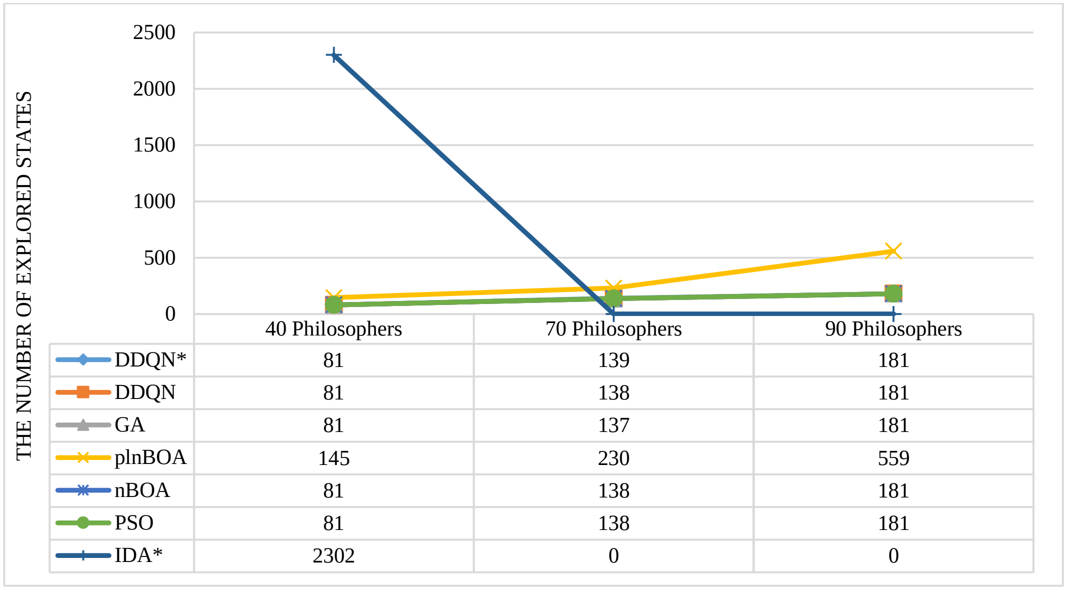
According to experiments, as shown in Fig. 26, in terms of length of generated witnesses, all approaches have the same results in 40 and 90 philosophers problems. However, in the 70 philosopher problem, the length of the generated witnesses in GA and plnBOA approaches was 136. For DDQN*, DDQN, nBOA and PSO approaches, this number was 138,137,137 and 137, respectively. Among the tested approaches, the GA was able to find the goal state at a lower depth and with fewer explored states than the others, according to Fig. 27.

In terms of overall performance, GA was the best. Given that the GA first traverses the path of a chromosome without any specific calculations, the speed of the operation in the GA makes it superior to other approaches. Since the general reward function is slightly more accurate than the dedicated reward, DDQN has a better performance in finding the shorter witness than the DDQN* approach. plnBOA explores more states than the others in the largest state space of this problem, i.e., 90 philosophers.



	40 Philosophers	70 Philosophers	90 Philosophers
DDQN*	80	138	180
DDQN	80	137	180
GA	80	136	180
plnBOA	80	136	180
nBOA	80	137	180
PSO	80	137	180
IDA*	80	0	0

Fig. 26. Comparing the length of the generated witnesses in the dining philosophers problem.



	40 Philosophers	70 Philosophers	90 Philosophers
DDQN*	81	139	181
DDQN	81	138	181
GA	81	137	181
plnBOA	145	230	559
nBOA	81	138	181
PSO	81	138	181
IDA*	2302	0	0

Fig. 27. Comparing the number of explored states in the dining philosophers problem.

Snake Problem: The host graph of the snake problem is designed in the GROOVE toolset and is shown in Fig. 28(a). Also, the goal state of this problem is shown in Fig. 28(b). In this model, every location is called a cell (of type node *n0*), and if a cell has *ispoint* label (of type edge *e0*) it means that the cell contains an apple. Given this figure, there should be no cell in the ground with an *ispoint* label. When the snake eats all the apples in the ground, the agent reaches the goal state.

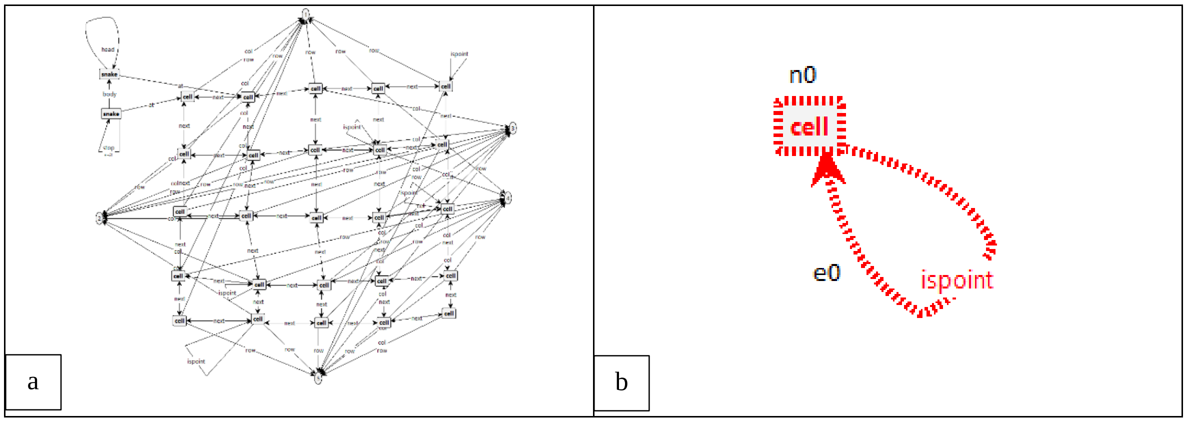
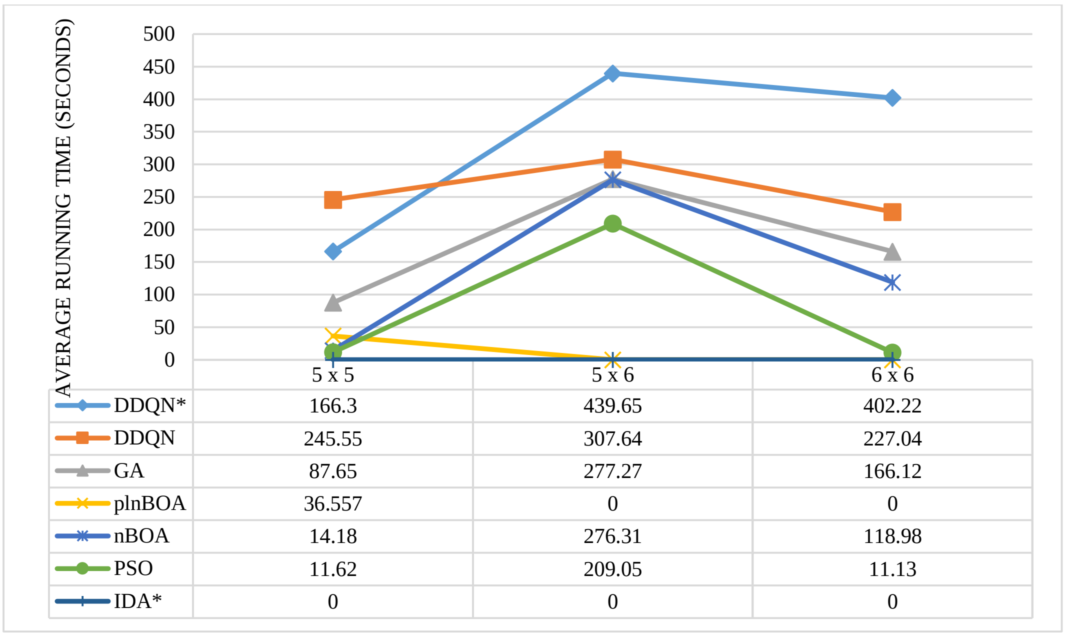


Fig. 28. A GTS-based model of the snake problem designed in the GROOVE toolset.

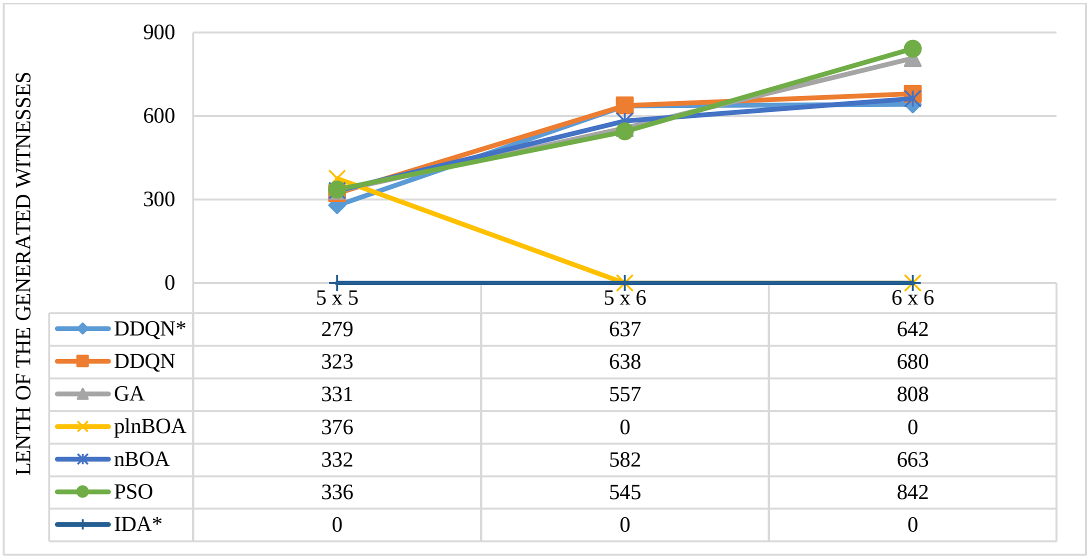
As a dedicated reward in DDQN*, the score of eating each apple is equal to 5, and other movements are equal to -1. Fig. 29 shows the comparison of the average running time in seconds, Fig. 30 shows the comparison of the length of the witnesses, Fig. 31 the comparison of the accuracy (i.e., the number of witnesses), and Fig. 32 shows the comparison of the number of explored states to satisfy the property in the snake problem.

According to Fig. 29 and Fig. 30, the proposed approach in most cases finds the goal state with optimal length, but due to its heavy calculations, it was not able to find the goal state in all runs in the limited time of each experiment. Because of this, the proposed approach shows that it needs more time than other approaches. The impact of the reward function is quite clear. The nBOA approach shows a remarkable performance due to the high dependency among the rules of this problem in which the new apple growth location rule is highly dependent on the previous apple growth location. However, the length of the witness is not optimal. The PSO was relatively fast, but it does not perform well enough in length and the number of witnesses by this approach. The GA is not very successful in terms of the length of the generated witness but shows that it can find the goal state more quickly and more likely than other approaches except nBOA.



	5 x 5	5 x 6	6 x 6
DDQN*	166.3	439.65	402.22
DDQN	245.55	307.64	227.04
GA	87.65	277.27	166.12
plnBOA	36.557	0	0
nBOA	14.18	276.31	118.98
PSO	11.62	209.05	11.13
IDA*	0	0	0

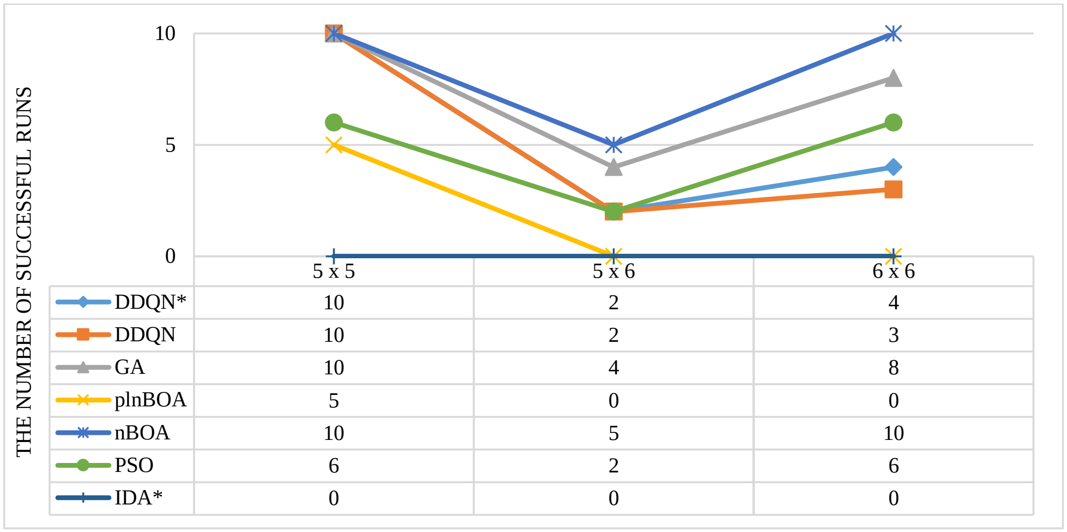
Fig. 29. Comparing the average running time in seconds in the snake problem.



	5 x 5	5 x 6	6 x 6
DDQN*	279	637	642
DDQN	323	638	680
GA	331	557	808
plnBOA	376	0	0
nBOA	332	582	663
PSO	336	545	842
IDA*	0	0	0

Fig. 30. Comparing the length of the generated witnesses in the snake problem.

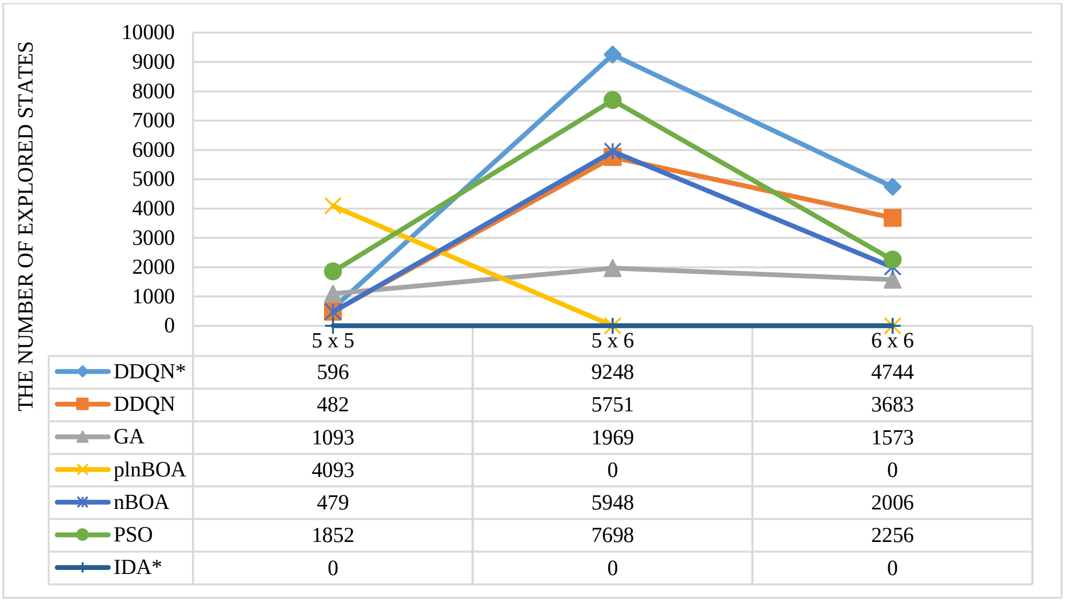
According to the results of Fig. 31, IDA* fails in this problem and also plnBOA provides a poor performance and just find the goal state in a 5 x 5 environment.



	5 x 5	5 x 6	6 x 6
DDQN*	10	2	4
DDQN	10	2	3
GA	10	4	8
plnBOA	5	0	0
nBOA	10	5	10
PSO	6	2	6
IDA*	0	0	0

Fig. 31. Comparing the ratio of the number of successful runs to the total runs in the snake problem.

Also, according to Fig. 32, the nBOA and GA approaches achieve the goal state by exploring fewer states, especially in the most complex environment. DDQN was better than DDQN*, again because of the more accurate reward function.



	5 x 5	5 x 6	6 x 6
DDQN*	596	9248	4744
DDQN	482	5751	3683
GA	1093	1969	1573
plnBOA	4093	0	0
nBOA	479	5948	2006
PSO	1852	7698	2256
IDA*	0	0	0

Fig. 32. Comparing the number of explored states in the snake problem.

5.5. Discussion

In this section, the results from 5.4. Evaluation is collected and compared to each other to find the advantages and disadvantages of the proposed approach. It should be noted that the statistics were extracted through the SPSS statistical program.

Fig. 33 shows the success rate of all approaches in all experiments. The proposed approach performs better than other approaches in terms of accuracy (i.e., the number of witnesses). However, due to excessive calculations and time limitations, the accuracy of the DDQN approach is reduced and comes in fourth place. It shows that the bottleneck of the proposed approach is the reward function and has a huge impact on the results. The proposed approach is 15% better than the runner up (i.e. GA), and on average, it is 37% better than the other approaches in terms of accuracy.

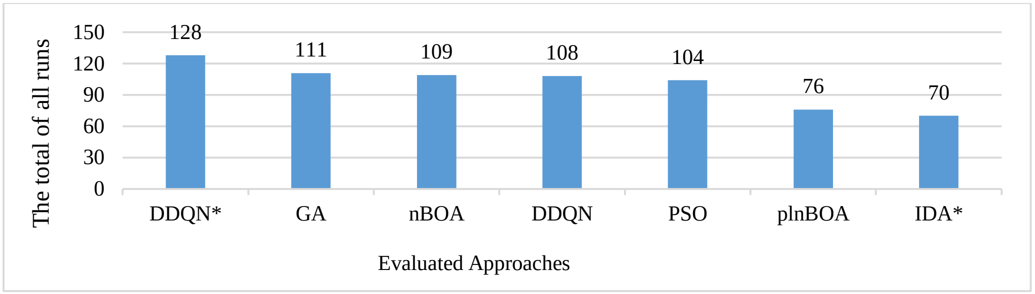


Fig. 33. Comparing the ratio of the number of successful runs to the total runs in all problems.

Because of the structure of the proposed approach in maintaining experiences, it is possible to provide better witnesses by rerunning the algorithm. In fact, in every run of the proposed approach, the last data is remembered and used to complete the dataset. The agent will go with more knowledge than the previous run in the next run. Fig. 34 shows the results of the sequential execution of the proposed approach in 100 seconds on the problem of dining philosophers with 70 philosophers. The length of the first generated witness is 139, and after five repetitions of the algorithm, this number is decreased to 135.

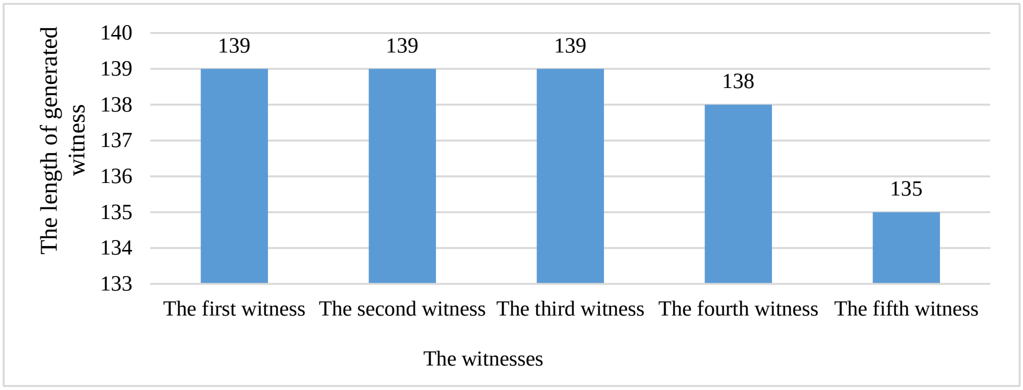


Fig. 34. Improving the length of the witness by sequentially implementing the proposed approach on the dining philosophers problem with 70 philosophers

The proposed approach provides shorter witnesses in some case studies such as 8-puzzle, N-Queen and dining philosophers. Table 3 fully addresses this issue. According to these results, in most case studies, the proposed approach with a dedicated reward has provided more optimal witnesses in terms of the length. Also, it is better than the general reward approach that again shows the importance of reward function. On average, the proposed approach is 300% better than the others in terms of generating shorter witnesses.

Table 3 Comparing the length of the generated witnesses in all problems.

Ranks	**DDQN* - DDQN**	**DDQN* - GA**	**DDQN* - plnBOA**	**DDQN* - nBOA**	**DDQN* - PSO**	**DDQN* - IDA***
The superiority of the DDQN*	8	8	9	7	6	11
The superiority of the other approaches	1	2	1	7	3	0
The number of draws	6	5	5	6	6	4

The reported results show that this approach searches smartly and explores fewer states than other approaches to reach the goal in most experiments. The relevant statistics are presented in **Error! Reference source not found.**. It is clear that the proposed approach has performed better in terms of efficiency exploring among the tested approaches. Our approach is 60% better than nBOA and 125% better than GA and PSO, and on average, it is 400% better than the others in terms of exploring fewer states.

Table 4 Comparing the number of explored states in all problems.

Ranks	**DDQN* - DDQN**	**DDQN* - GA**	**DDQN* - plnBOA**	**DDQN* - nBOA**	**DDQN* - PSO**	**DDQN* - IDA***
The superiority of the DDQN*	5	9	14	8	9	15
The superiority of the other approaches	8	4	1	5	4	0
The number of draws	2	2	0	2	2	0

The results obtained from the average running time from all experiments are evaluated by the Wilcoxon signed-rank test [32] and provided in

Table 5. In the Wilcoxon signed-rank test, if the decision criterion (sig.) is less than 0.05, it shows that there is a significant difference between two groups of data, i.e. comparison of the average running time of two different approaches. Also, in this test, it is assumed that a positive rank means DDQN* approach is faster than the other approach. Otherwise, it is slower. As it turns out, the proposed approach with a dedicated reward is significantly faster than the general reward one and also the IDA* approach. However, the opposite is true for comparison with the nBOA approach. According to the results of

Table 5, where the value of the signed number of standard deviations (Z) comparing the DDQN* and nBOA approaches based on positive ranks, it shows that nBOA approach is much faster. However, the proposed approach is slightly faster than plnBOA. In the case of other approaches, the overall speed has not changed at all.

Table 5 Results of the Wilcoxon signed-rank test.

	DDQN*- DDQN	**DDQN*-GA**	**DDQN*-plnBOA**	**DDQN*-nBOA**	**DDQN*- PSO**	**DDQN*-IDA***
Z	-2.158[a]	-0.057[b]	-0.057[a]	-1.988[b]	0	-2.613[a]
Sig.	0.031	0.955	0.307	0.047	1.0	0.009

a. Based on positive ranks
b. Based on negative ranks

6. Conclusion and future works

Model-checking is one of the well-known and formal techniques in the verification of software systems and is used to check some properties such as reachability. The goal of satisfying reachability property is to find the desired state in the state space. However, applying this technique may lead to the state space explosion problem in which all reachable states cannot be checked due to computational limitations. Some approaches such as GA and PSO have been proposed to handle this problem. These approaches intelligently explore only a portion of the state space instead of an exhaustive exploration. The proposed approach in this paper is based on deep reinforcement learning and can explore the state space by using machine learning and also interacting with the environment. In this way, the agent will perform an action and wait for the environment to give it a reward or penalty. Afterwards, the agent will remember the performed action and its result and try to perform better action based on its experiences. In this paper, two types of rewards are given to the agent: (1) Dedicated reward for each problem and (2) General reward for all problems. The proposed approach proves that it has good potential in providing optimal performance in terms of shorter generated witnesses, fewer explored states and higher accuracy in finding the goal state than other approaches. Also, changing the rewards for each problem causes the agent's behaviour changes in the environment to search for a goal state. Eventually, the proposed approach proves that it can improve the generated witness after the sequential execution of the algorithm.

Advantages of the proposed approach:

From the results, these advantages can be concluded:

1. The proposed approach performs better than other case studies in terms of accuracy
2. Can provide shorter witnesses at first and also after sequentially running of the algorithm
3. Searches smartly and explores fewer states than other approaches
4. The user can quickly solve the desired problem by designing and modifying the dedicated reward function. The different results from the proposed approach with the general reward and the dedicated reward prove this.

Limitations of the proposed approach:

Due to the many calculations of the proposed approach to get the reward's value in each state, other approaches can find the goal state faster. However, their generated witness is not necessarily optimized. We should remember that the time factor cannot be ignored, and our resources are limited. Reward function has an important role in the proposed approach and is used as a guide for an agent. So, eliminating the reward function is inevitable. However, it can be optimized.

Recommendations:

Future works can be considered as follows:

- There are many approaches, such as Actor-Critic and Policy Gradient, in the field of reinforcement learning that can be implemented. The Policy Gradient method calculates the reward at the end of each path. Thus, it is no need to calculate the reward on each step.

- Instead of randomly selecting rules at the beginning of the search process to complete the dataset for the neural network, more efficient methods can be used to build this dataset.
- It is also possible to optimize the general reward function so that the calculations of this function are more accurate and faster than before.
- Instead of a random selection, the experience batch can be fetched more intelligently from experience replay memory to train the neural network, which results in more accuracy.

References

[1] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[2] S. Demri, F. Laroussinie, and P. Schnoebelen, "A parametric analysis of the state-explosion problem in model checking," *Computer and System Sciences*, vol. 72, no. 4, pp. 547-575, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000005001297.

[3] R. Yousefian, V. Rafe, and M. Rahmani, "A heuristic solution for model checking graph transformation systems," *Applied Soft Computing*, vol. 24, pp. 169-180, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1568494614003305.

[4] E. Snippe, "Using heuristic search to solve planning problems in GROOVE," in *14th Twente Student Conference on IT, University of Twente*, 2011. [Online]. Available: https://www.researchgate.net/publication/228977418_Using_Heuristic_Search_to_Solve_Planning_Problems_in_GROOVE. [Online]. Available: https://www.researchgate.net/publication/228977418_Using_Heuristic_Search_to_Solve_Planning_Problems_in_GROOVE

[5] E. Pira, V. Rafe, and A. Nikanjam, "Deadlock detection in complex software systems specified through graph transformation using Bayesian optimization algorithm," *Systems and Software*, vol. 131, pp. 181-200, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217301061.

[6] G. Rozenberg, *Handbook of Graph Grammars and Comp*. World scientific, 1997.

[7] A. Rensink, "The GROOVE simulator: A tool for state space generation," in *International Workshop on Applications of Graph Transformations with Industrial Relevance*, 2003: Springer, pp. 479-485. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-25959-6_40. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-25959-6_40

[8] J. Maeoka, Y. Tanabe, and F. Ishikawa, "Depth-first heuristic search for software model checking," in *Computer and Information Science 2015*: Springer, 2016, pp. 75-96.

[9] R. Yousefian, S. Aboutorabi, and V. Rafe, "A greedy algorithm versus metaheuristic solutions to deadlock detection in Graph Transformation Systems," *Intelligent and Fuzzy Systems*, vol. 31, no. 1, pp. 137-149, 2016. [Online]. Available: https://www.researchgate.net/publication/301542900_A_greedy_algorithm_versus_metaheuristic_solutions_to_deadlock_detection_in_Graph_Transformation_Systems.

[10] V. Rafe, M. Darghayedi, and E. Pira, "MS-ACO: a multi-stage ant colony optimization to refute complex software systems specified through graph transformation," *Soft Computing*, vol. 23, no. 12, pp. 4531-4556, 2019.

[11] E. Pira, V. Rafe, and A. Nikanjam, "Using evolutionary algorithms for reachability analysis of complex software systems specified through graph transformation," *Reliability Engineering & System Safety*, vol. 191, p. 106577, 2019.

[12] E. Pira, V. Rafe, and A. Nikanjam, "Searching for violation of safety and liveness properties using knowledge discovery in complex systems specified through graph transformations," *Information and Software Technology*, vol. 97, pp. 110-134, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S095058491830003X.

[13] E. Pira, "A novel approach to solve AI planning problems in graph transformations," *Engineering Applications of Artificial Intelligence*, vol. 92, p. 103684, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0952197620301214.

[14] E. Pira, V. Rafe, and A. Nikanjam, "EMCDM: Efficient model checking by data mining for verification of complex software systems specified through architectural styles," *Applied Soft Computing*, vol. 49, pp. 1185-1201, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1568494616303192.

[15] J. Partabian, V. Rafe, H. Parvin, and S. Nejatian, "An approach based on knowledge exploration for state space management in checking reachability of complex software systems," *Soft Computing*, vol. 24, no. 10, pp. 7181-7196, 2020.

[16] M. Yasrebi, V. Rafe, and S. Nejatian, "An efficient approach to state space management in model checking of complex software systems using machine learning techniques," *Journal of Intelligent & Fuzzy Systems*, vol. 38, no. 2, pp. 1761-1773, 2020.

[17] E. Pira, "Using knowledge discovery to propose a two-phase model checking for safety analysis of graph transformations," *Software Quality Journal*, pp. 1-28, 2021.

[18] N. Jansen, B. Könighofer, S. Junges, and R. Bloem, "Shielded decision-making in MDPs," *arXiv.org e-Print archive*, vol. abs/1807.06096, 2018. [Online]. Available: https://www.researchgate.net/profile/Nils_Jansen/publication/326459531_Shielded_Decision-Making_in_MDPs/links/5be0224e4585150b2b9faeed/Shielded-Decision-Making-in-MDPs.pdf?origin=publication_detail.

[19] N. Fulton and A. Platzer, "Safe reinforcement learning via formal methods: Toward safe control through proof and learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/download/17376/16225. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/download/17376/16225

[20] T. Liu, B. Tian, Y. Ai, L. Li, D. Cao, and F.-Y. Wang, "Parallel reinforcement learning: A framework and case study," *IEEE/CAA Journal of Automatica Sinica*, vol. 5, no. 4, pp. 827-835, 2018.

[21] A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro, "Translating Java code to graph transformation systems," in *International Conference on Graph Transformation*, 2004: Springer, pp. 383-398. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-30203-2_27. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-30203-2_27

[22] G. Taentzer, "AGG: A graph transformation environment for modeling and validation of software," in *International Workshop on Applications of Graph Transformations with Industrial Relevance*, 2003: Springer, pp. 446-453. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-25959-6_35. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-25959-6_35

[23] J. De Lara and H. Vangheluwe, "AToM 3: A Tool for Multi-formalism and Meta-modelling," in *International Conference on Fundamental Approaches to Software Engineering*, 2002: Springer, pp. 174-188. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45923-5_12. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45923-5_12

[24] G. Bergmann *et al.*, "Viatra 3: A reactive model transformation platform," in *International Conference on Theory and Practice of Model Transformations*, 2015: Springer, pp. 101-110. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-21155-8_8. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-21155-8_8

[25] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[26] I. A. Hosu and T. Rebedea, "Playing atari games with deep reinforcement learning and human checkpoint replay," *arXiv.org e-Print archive*, 2016. [Online]. Available: https://arxiv.org/abs/1607.05077.

[27] D. P. Bertsekas, "Feature-based aggregation and deep reinforcement learning: A survey and some new implementations," *IEEE/CAA Journal of Automatica Sinica*, vol. 6, no. 1, pp. 1-31, 2018.

[28] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI conference on artificial intelligence*, 2016. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12389/11847. [Online]. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/download/12389/11847

[29] P. Duboue, *The Art of Feature Engineering: Essentials for Machine Learning*. Cambridge University Press, 2020.

[30] V. Rafe, M. Moradi, R. Yousefian, and A. Nikanjam, "A meta-heuristic solution for automated refutation of complex software systems specified through graph transformations," *Applied Soft Computing*, vol. 33, pp. 136-149, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1568494615002598.

[31] M. Amiri, H. B. Amnieh, M. Hasanipanah, and L. M. Khanli, "A new combination of artificial neural network and K-nearest neighbors models to predict blast-induced ground vibration and air-overpressure," *Engineering with Computers*, vol. 32, no. 4, pp. 631-644, 2016. [Online]. Available: https://link.springer.com/article/10.1007/s00366-016-0442-5.

[32] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*: Springer, 1992, pp. 196-202.