Received 22 November 2023, accepted 15 December 2023, date of publication 25 December 2023,
date of current version 3 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3346908

RESEARCH ARTICLE

Q-Learning Based Cognitive Domain Ontology Representation and Solving on Low Power Computing Platforms

NAYIM RAHMAN[1], (Member, IEEE), TANVIR ATAHARY[1], CHRIS YAKOPCIC[1], (Senior Member, IEEE), TAREK M. TAHA[1], (Senior Member, IEEE), AND SCOTT DOUGLASS[2]
[1]Department of Electrical and Computer Engineering (ECE), University of Dayton, Dayton, OH 45469, USA
[2]Air Force Research Laboratory, Wright Patterson AFB, Dayton, OH 45433, USA

Corresponding author: Nayim Rahman (rahmanm12@udayton.edu)

This work was supported by Air Force Research Laboratory (AFRL) under Grant AFRL-2021-4287.

ABSTRACT Cognitive agents make systems autonomous through the process of decision automation by mining an existing knowledge repository at run time. These processes can often be highly compute intensive, and would thus run slowly on the low-power computing platforms typically seen in autonomous systems. This paper examines how knowledge be represented in a Q-table and proposes a novel fast algorithm to mine that knowledge based on constraints. We evaluate this approach for the knowledge mining process of a specific agent: Cognitively Enhanced Complex Event Processing (CECEP). Within CECEP, knowledge is represented using Cognitive Domain Ontologies (CDO), and is mined using situational inputs and constraints. This is a novel approach to store information and is able to accommodate CDOs with millions of solutions. To show that the approach can run on low power hardware in real-time, this algorithm was executed on two low-power minicomputing platforms - Intel's NUC and Asus's Tinker Board. At present, no other optimized CDO solvers can generate solutions on these platforms. The algorithm generated the same amount of solutions as a GPU-enabled optimized path-based forward checking CDO solver, while consuming around 7.7 and 5.15 times less energy (Joules) on the NUC and Tinker Board respectively.

INDEX TERMS Knowledge mining, cognitive agents, autonomous decision making.

I. INTRODUCTION
Autonomous systems are in high demand because of their presence in a variety of domains, such as UAVs (Unmanned Aerial Vehicle), self-driving cars, robots, planning, data mining, and operations research. A cognitive agent within an autonomous system makes decisions based on its surrounding environmental inputs and domain knowledge. Several cognitive architectures have been developed [1], [2], [3], [4], [5], [6], over time, of which SOAR [3] and ACT-R [4], [5], [6], are the most widely explored. Cognitive scientists have developed a Cognitively Enhanced Complex Event Processing (CECEP) architecture [7], [8], [9], [10], by combing complex event processing and cognitive modeling for enhanced reasoning and complex decision making in a variety of domains.

The CECEP framework consists of a group of net-centric event processing components is capable of processing declarative, procedural, and domain knowledge. Within the CECEP architecture, soaCDO is the knowledge representation and mining component. Knowledge mining within soaCDO is the most time-consuming and power-hungry task in the CECEP architecture. Domain knowledge within a Cognitive Domain Ontology (CDO) makes a cognitive agent capable of operating autonomously in multiple contexts using a set of complex constraints. A CDO can generate either a specific solution or a group of solutions based on its environmental inputs and domain knowledge.

The associate editor coordinating the review of this manuscript and approving it for publication was Angelo Trotta.

There is a strong relationship between the scalability and complexity in these problems with the power and time required to solve them. Consequently, it is difficult for the CECEP framework to evaluate complex real-world scenarios in real-time on power-constrained autonomous systems. The solution search space of a CDO can be enormous (over 10^{30} solutions for some of the problems in this study), making it difficult for existing CDO solvers to perform in real-time without consuming much power. Among all the existing CDO solving approaches, only a GPU-enabled highly parallel approach [11] can solve large-scale CDOs in run-time. The power and memory requirement of this approach makes it unfeasible for low power platforms. Thus, there is a need for new approaches to solve CDOs on low-power systems at run-time.

This work examines novel approaches for data representation and mining within CDOs using Q-learning to enable real-time execution on low-power systems. We utilize Q-learning to map a CDO's knowledge into tabular form and then mine that knowledge in real-time using a novel extraction algorithm [12]. The Q-learning algorithm is primarily used where agents interact with their environment and act based on feedback. It is utilized in various domains to optimize outcomes and has application in many fields, including optimal path planning for mobile robots [13], [14], [15], optimal channel and power allocation for a network [16], [17], urban water resource management [18], and load balancing of supplier-agents for the electricity market [19].

The key contribution of this article is a knowledge representation and mining approach that is developed using the Q-learning algorithm. This is a novel concept in both operations research and artificial intelligence. Our main contributions are summarized as follows:

- We represent knowledge using the strategy of the shortest pathfinding problem. In the shortest pathfinding problem, the shortest path is mapped in a Q-table using Bellman's equation. In this study, the entire domain knowledge of a CDO is converted to an equivalent set of paths that are stored within a Q-table.
- At run-time a novel knowledge extraction algorithm is used to mine the knowledge from the Q-table. This algorithm can traverse all directions within the Q-table to extract knowledge. This new approach does not require any specific starting point to generate a complete solution. Any intermediate state (knowledge element) of an entire path can be used as a starting point.

We ensure that the solutions are successfully mapped in a Q-table using the shortest path finding approach by validating with existing CDO solving approaches. Both the knowledge mapping and extraction process are explained in later sections. As the CDO knowledge and constraints are stored within a Q-table during the training phase, a user can easily understand why a particular input event generates a specific set of solutions. The explainable feature of this method makes the approach reliable. This proposed approach can handle large search spaces in runtime. It can generate solutions for complex CDOs while consuming significantly less power compared to existing optimized CDO solvers.

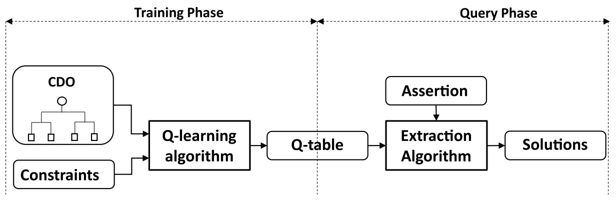


FIGURE 1. Schematic diagram of the knowledge mapping and extraction process.

The approach presented in this paper could potentially be used in memory modules of other cognitive architectures with proper mapping. Different cognitive architectures use different structures of domain knowledge to operate in real-time. ACT-R uses two kinds of memory modules (declarative and procedural memory [20]), while SOAR uses a combination of current sensory data, prior knowledge about solving problems, and long-term relevant memory to operate in real-time [21]. As cognitive architectures, such as SOAR, have applications in many areas including puzzles, games, simulated pilots, and autonomous robotics systems [22], the approach in this paper could be useful for cognitive systems in many areas.

The overall knowledge mapping and extraction process is shown in Fig. 1. The CDO and constraints are first mapped in a Q-table using the Q-learning algorithm efficiently during the training phase. The agent will then generate solutions from the Q-table using the extraction algorithm based on assertion during runtime.

Three different configurations of a CDO are tested to prove our method's validity and scalability. Eventually, the Q-learning-based CDO solving approach is compared against a highly optimized GPU enabled CDO solver [11] to prove the capability and efficiency of the developed method on low power platforms.

The experiments in this study are conducted on low power computing devices, including the Asus Tinker Board and the Intel NUC. The power consumption of these devices is far less than the power consumed by multi-core and GPU based systems that are traditionally used to mine knowledge. Our proposed approach generated millions of solutions, all while consuming around 7.7 and 5.15 times less energy than the GPU-enabled optimized solver, running on the Intel NUC and the Asus Tinker Board respectively.

The rest of the paper is organized as follows: the CECEP and CDO architectures are briefly introduced in Sections II and III respectively. Related works are discussed in Section IV. Q-learning is discussed in Section V, while the CDO mapping strategy using Q-learning and knowledge query are discussed in Section VI. Experimental setup and results are discussed in Sections VII and VIII. Section IX concludes the paper.

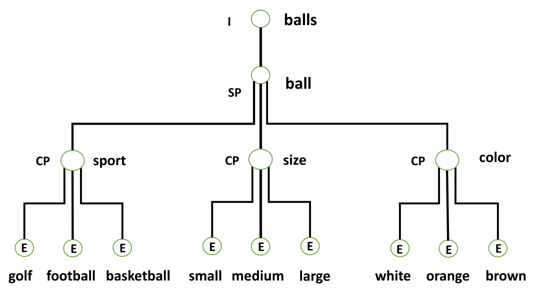


FIGURE 2. Domain structure of ball CDO.

II. THE CECEP ARCHITECTURE

The CECEP framework is based on a net-centric architecture that has been developed for agents specified in a collection of agent-specification formalisms. The Research Modeling Language (RML) is a domain-specific language that has been designed to enable the rapid development and deployment of intelligent and autonomous agents [8], [9], [10]. RML requires a user to represent the models and agents as complex event processing agents through events, event patterns, event pattern rules, behavior models, and cognitive domain ontologies to maximize the scalability and interoperability during execution and simulation.

The CECEP architecture currently consists of the following central net-centric components:

soaDM: an associative memory application that allows RML models and agents to store and retrieve *declarative knowledge* [9].

soaCDO: a knowledge representation and mining application that allows RML models and agents to save and exploit *domain knowledge*.

Esper: a process that provides a complex event processing framework that allows RML models and agents to perform actions on context assessment and *procedural knowledge* [10].

Through these components, the CECEP architecture incorporates model and agent capabilities based on *declarative*, *procedural*, and *domain* knowledge processing to the Esper framework. Although a detailed description of CECEP is beyond the scope of this paper, we do describe the CDO component below.

III. COGNITIVE DOMAIN ONTOLOGIES

Cognitive Domain Ontologies (CDOs) formalize the CECEP Agent's domain knowledge. A CDO represents the structure of a domain and the relationships among its components. Formally, a CDO is a tree with alternating entities and relations. Entities correspond to domain objects, such as a playing card's value or suit. A CDO consists of three major entities: SubParts (SP), ChoicePoints (CP), and Instances (I = [0...n]).

A. CDO STRUCTURE

Let's, consider a simple Ball CDO to understand the knowledge representation, its complexity, and the solution generation process. The Ball CDO in Fig. 2 represents different ball related knowledge, including sport, size, and color in a hierarchical structure.

TABLE 1. User-defined constraints for the ball CDO.

Specification (User Defined Constraints)
If sport is golf, then size is small, and color is white.
If sport is football, then size is medium, and color is brown.
If sport is basketball, then size is large, and color is orange.

In a CDO, SubParts contain a unification relationship with other entities, where all entities necessarily occur together (i.e., the 'ball' SubParts node in Fig. 2 indicates that '*sport*', 'size', and 'color' events must occur together in the solution).

ChoicePoints represent an "either-or" relationship among entities; only one may be active at a time. For example, the 'sport' ChoicePoint node in Fig. 2 can be either golf, football, or basketball. Instances capture the replicated sub-structure. The CDO in Fig. 2 does not include an example of this type of structural relationship, but the CDO in Fig. 3 does. A detailed explanation of the instance entity is discussed in the following section.

Entities could have zero or more event attributes. User-defined constraint relationships allow users to connect events and attributes in a CDO to each other using conditionals (*if, iff*), connectives (and, or, not), and attribute comparisons (<, >, <=, >=, !=, etc.). The three constraints applicable to the Ball CDO are listed in Table 1. The combination of structural domain knowledge (see Fig. 2) and relational domain knowledge (see Table 1) yields a complete CDO.

In Table 1, each constraint is true for all the permutations of the events of the same constraint. This property of the constraints is known as the integrity of constraints. For example, the first constraint will create the following equally true constraints:

1) *If size is small, then sport is golf, and color is white*
2) *If color is white, then sport is golf, and size is small.*
3) *If sport is golf, and size is small, then color is white.*
4) *If sport is golf, and color is white, then size is small.*
5) *If color is white, and size is small, then sport is golf.*

In the Ball CDO in Fig. 2, each of the ChoicePoints contains three entities where each entity is an "either-or" relationship with other entities within their domain (i.e., sport, size, and color). A CDO generates its solution by combining the knowledge from all of the SubParts. If there are no user-defined constraints for the CDO, then any combination of these ChoicePoints will be a solution of the CDO. There are 27 possible solutions available for the Ball CDO without constraints. Two example solutions are: 'golf-small-white', and 'golf-medium-brown'.

User-defined constraints restrict valid solutions to a subset of all possible combinations. Thus, the number of valid solutions for a given scenario varies based on the constraints. The size of the solution space also depends on the complexity of the constraints. If the constraints from Table 1 are considered, the solution space reduces from twenty-seven to three:

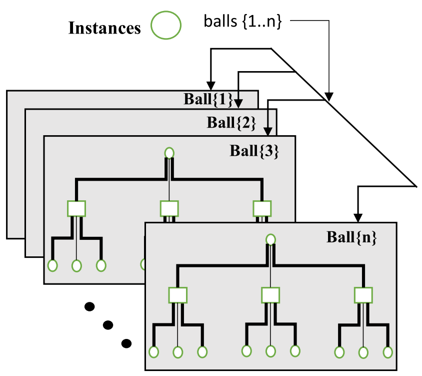


FIGURE 3. Structural representation of the ball CDO for n instances.

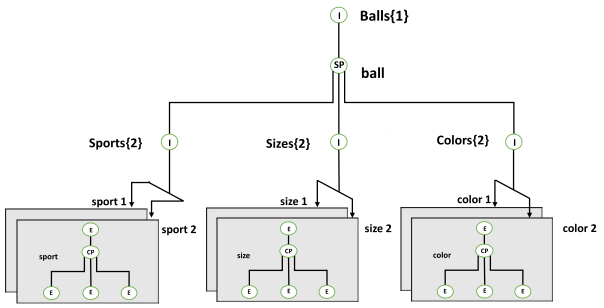


FIGURE 4. Structural representation of Ball CDO with the domain instance.

1) *Sport(golf) – Size(small) – Color(white)*
2) *Sport(football) – Size(medium) – Color(brown)*
3) *Sport(basketball) – Size(large) – Color(orange)*

B. MULTI-INSTANCE CDO (CDO WITH INSTANCES)

The root node (topmost node) of this Ball CDO is an instance node in Fig 2. This is denoted by the letter "I" next to this root node. An instance node defines the replication of the remaining CDO structure [11]. Fig. 3 displays a representation of the Ball CDO, where the 'balls' instance node is set to *n*. The number of solutions grows when the instance number is set to a higher value.

Instance nodes can be added to any branch of a CDO. Also, different constraints can be applied at different instance levels. Let's assume three more instances are added to the Ball CDO at the domain level, e.g., sports, sizes, and colors. If each of these new domain-level instances (sports{2}, sizes{2}, and colors{2}) are set to 2, and the balls instance node is set to 1, then this Ball CDO will appear as shown in Fig. 4.

Each instance contains partial solutions which must be combined to form a complete solution. The six instance-level solutions, or partial solutions available for the example ball CDO are as follows: *'Golf{1} Small{1} White{1}', 'Football{1} Medium{1} Brown{1}', 'Basketball{1} Large {1} Orange {1}', 'Golf{2} Small{2} White{2}', Football{2} Medium{2} Brown{2}', and 'Basketball{2} Large{2} Orange{2}'* (see Fig. 4).

As each partial solution set contains three solutions, the total number of complete solutions will be calculated by

TABLE 2. Two complete solutions for the ball cdo of fig 4.

Solution No	Complete Solution
1	Golf{1}, Small{1}, White{1} Golf{2}, Small{2}, White{2}
2	Football{1}, Medium{1}, Brown{1} Basketball{2}, Large{2}, Orange{2}

TABLE 3. Solution space for different instance settings.

Instances	Search space without constraints	Search space with Constraints
balls{1}, sports{1}, sizes{1}, colors{1}	27	3
balls{1}, sports{2}, sizes{2}, colors{2}	729	9
balls{2}, sports{2}, sizes{2}, colors{2}	531441	81

Here, solution space refers to the number of possible solutions to the Ball CDO with and without Constraints for Different Instance Settings.

multiplying the number of partial solutions ($3 \times 3 = 9$) as shown in (1). Here '||' represents the concatenation of the partial solutions.

$$\text{balls}\{1\} => (\text{sport1}, \text{size1}, \text{color1})_{\text{instance1}}|| \\ (\text{sport2}, \text{size2}, \text{color2})_{\text{instance2}} \quad (1)$$

A simplified representation of two complete solutions is shown in Table 2. The total number of solutions is highly dependent on values of the instance nodes and the complexity of the constraints. For higher instance values, the possible number of solution combinations is higher, and thus the number of possible complete solutions becomes larger.

Constraint satisfying solutions are members of the entire solution space. Due to the constraints, solvers need to traverse the entire solution space to determine which solutions are valid. In many application scenarios (such as mobile platforms), this task needs to be done fast and at low power.

Table 3 shows the solution search space to the Ball CDO (see Fig. 4) with and without constraints for different instance settings. The same constraints (see Table 1) are applied to all instances. As the instance values increase, the number of valid solutions also increases for the same CDO.

The solution search space increases exponentially with the increase in instances. This search space can easily exceed one nonillion (10^{30}) with the slight increase in instances for a moderate-sized CDO (see Table 10). Only highly optimized solvers can solve this problem at runtime.

CECEP agents mine knowledge from CDOs before making a decision. As a CDO holds structural and relational domain knowledge about a specific domain, every CDO is unique based on its domain. A CDO can generate different results based on the constraints applied, as the constraints define relations among the entities.

IV. RELATED WORKS

A. HIGH PERFORMANCE COMPUTING CDO SOLVING APPROACHES

Several algorithmic approaches have examined how to solve CDOs using high-performance computing systems to obtain all solutions to a problem. They solve larger problems very fast but require a tremendous amount of computing power (such as a GPU or multi-processor system). In [7], an exhaustive depth-first search based algorithm was developed to solve large-scale CDOs, where a CDO was first converted into an equivalent constraint satisfaction problem (CSP). A parallel version of that algorithm executed on a system with an Intel Xeon processor and a NVIDIA Tesla C2070 GPU is about 1000 times faster than the serial version. The latter was about 30% faster than using screamer + to solve the CDO. Screamer+ [23] is a LISP based non-deterministic constraint programming environment. In Screamer+, problems along with constraints are expressed in LISP lists, and the Screamer+ algorithm was used to solve CDOs. A modified version of that exhaustive depth-first search algorithm [24] implemented with a NVIDIA graphics processor (Tesla C2070) achieved 100 times speedup over a Xeon processor implementation and almost 8 times speedup over a Xeon Phi processor implementation.

Forward checking prunes the solution search space based on the constrains provided and as such is more efficient than exhaustive depth first search brute force approaches. A serial forward checking CDO solver [25] achieved around 10-25 times speedup to generate the first solution and achieved around half a million times speedup for all solutions compared to a java based CDO solver called Sherlock (based on the Choco 2.15 constraint solver library [26]). The parallel version of that path-based forward checking algorithm was examined on 128 compute nodes at the Ohio Supercomputing Center and achieved 200 times speedup compared to the serial version [11]. A solution ranking capable CDO solver [27] was developed, which was able to rank its solution using various optimization functions. This utilized the forward checking-based algorithm to generate all solutions on a CPU, and these solutions were then ranked using an algorithm running on a GPU.

The cognitive architecture ACT-R's memory module, soaDM (service-oriented architecture Declarative Memory) was accelerated with GPUs. This was referred to as a Hardware Accelerated Declarative Memory (HADM) system [28]. It achieved approximately 5 times speedup over the previously used Accelerated Declarative Memory (ADM) system [29]. The soaDM memory module is also utilized in the CECEP architecture as an associative memory application.

B. LOW POWER CDO SOLVING APPROACHES

Apart from conventional approaches, different low power neuromorphic device-based approaches have been examined. A convolution neural network-based approach used IBM's TrueNorth neuromorphic device and the "Eedn" framework to solve small to mid-sized CDOs [12], [30]. The power consumption for that approach was around 50mW. A lookup table-based approach was used to solve CDOs [31] on IBM's TrueNorth neuromorphic device and also consumed around 50mW for small-sized CDOs. Although neuromorphic methods require many cores to map larger problems, they consume very little power.

C. KNOWLEDGE MINING APPROACHES

There are several studies about knowledge extraction and manipulation processes for different domains. In [32], a combined mining approach was proposed for enterprise applications, such as telecom fraud detection. This approach extracted information from complex data and is capable of handling multi-feature, multi-source, and multi-method related issues. In [33], a knowledge fusing approach was discussed, where models trained from the sample data were fused in the level of the parameters to have a combined model for a distributed system. In that approach, various distributed agents collected sample data from the environment, and that combined model was used to extract knowledge from complex data. In [34], the possibility of using artificial neural networks such as Self-Organizing Maps (SOM), Neuro-Fuzzy, ART2, and Backpropagation in data mining was discussed. In [35], the author proposed a domain-driven actionable knowledge delivery concept instead of data-centered knowledge discovery to solve real-world knowledge mining issues by incorporating Ubiquitous Intelligence. Some of the components of Ubiquitous Intelligence are domain knowledge/intelligence, data intelligence, and network intelligence, which need to be incorporated into the solving mechanism.

D. Q-LEARNING APPLICATIONS

Q-learning is typically used in optimal decision-making problems where agents continuously interact with the environment. An improved version of the Q-learning algorithm based on the flower pollination algorithm (FPA) was used to plan an optimal path for a mobile robot in a static environment [13]. For the dynamic path planning of a mobile robot, Q-learning was incorporated with some heuristic-based searching strategies to reduce the search space [14]. That approach converged quickly compared to classical Q-learning. In another application to the mobile robot path planning problem, a feed-forward Artificial Neural Network (ANN) based controller was developed where Q-learning was used to collect training samples [15]. A combination of ANN and Q-learning achieved better performance compared to only one of the two approaches. Q-learning was used in femtocell networks to learn optimal channel and power allocation [16]. Femtocell is a next-generation wireless network used to increase indoor coverage capacity. In [17],

Q-learning was used to solve the dynamic channel assignment problem for mobile communications considering homogenous and inhomogeneous traffic distributions, time-varying traffic patterns, and channel failures. Some other

Algorithm 1 Q-learning Algorithm (Training Phase)

```
Initialization:
  1. Define the reward table for each $(s, a)$ pair.
  2. Initialize the Q-table for each pair $(s, a)$ with zero.
  3. Set $\gamma$ value.
Steps:
  4. Observe the current state $s$.
  5. REPEAT:
     - Select an action $a$, randomly.
     - Receive immediate reward $r(s, a)$ from the reward table.
     - Observe the new state $s_n$ based on the previous action.
     - Update the table entry for $Q(s, a)$ as follows:
       $Q(s, a) = r(s, a) + \gamma * (\max(Q(s_n, a_n)))$
     - $s \leftarrow s_n$
```

dynamic agent-environment interactive resource management applications of Q-learning are the urban water resource management system [18] and the optimal supplier-agent system for the electricity market [19].

V. Q-LEARNING

Q-learning is an off-policy method of Reinforcement Learning that learns an optimal action-selection policy for any given finite Markov Decision Process. Q-learning is also known as a model-free learning algorithm, as it can learn the optimal policy without being aware of the model of the environment [36]. In Q-learning, an agent is only aware of the available states, the possible actions that can be taken from the current position, and the associated rewards for each possible action. The agent learns an optimal policy by iteratively interacting with the environment and updating itself. The letter Q describes the quality of the action taken in a specific state. Q-learning tries to learn an optimal policy that maximizes a global reward by iterating through Bellman's equation (2).

$$Q(s, a) = r(s, a) + \gamma * (\max(Q(s_n, a_n))) \tag{2}$$

Here $Q(s, a)$ is an estimated utility function, and the goal is to maximize this utility function by choosing a specific action a, at step s. The quantity $r(s, a)$ is the immediate reward for taking a certain action a at step s. The value γ is the discount factor that helps to generate a discounted expected reward for the best future action to be taken at the next state (s_n). The value γ upholds the balance between immediate and future rewards. Typically, the value of γ ranges from 0 to 1. Actions a,and an are taken at states s,and s_n, respectively. The best future action is determined using a maximum function, which helps to optimize the utility function greedily.

The Q-table will be updated with the Q-values $Q(s, a)$ after each episode. An episode is completed whenever an agent reaches the endpoint of a decision process. The model converges after an appropriate amount of training using Algorithm 1, and the resulting Q-table holds optimal values.

TABLE 4. Unique number representation of constraints.

Constraints	Number Representations
Small, White, Golf	(0, 3, 6)
Medium, Brown, Football	(1, 4, 7)
Large, Orange, Basketball	(2, 5, 8)

The optimal policy will be generated from the Q-table using a maximum function to select the best action for the current state during the testing phase.

VI. CDO MAPPING TO Q-TABLE

The proposed Q-learning approach can generate all possible solutions to a problem. However, due to memory limitations, a limited number of solutions are generated for higher-ordered problems. There are two main challenges when mapping a CDO to a Q-table. The first challenge is how to map a CDO along with its constraints to a Q-table during the training phase. The second is how to extract information based on assertions from the Q-table during the query phase. The query phase is similar to the testing phase of the traditional Q-learning approach, where solutions are generated using the Q-table. In our application, this phase generates results based on assertions.

In this approach, knowledge is represented as the shortest pathfinding problem. Traditionally, in the shortest pathfinding problem, the Q-learning algorithm is used to learn the best possible action required from each step to achieve the optimal path. The agent updates the Q-table using Bellman's equation, where only the best action will have the highest values. In the developed knowledge mapping approach, all the constraint satisfying solutions are considered a set of shortest paths. All those paths are mapped compactly in a Q-table efficiently. The knowledge conversion process and mapping in the Q-table are discussed in the 'Training Phase' sub-section. The knowledge extraction from the Q-table is discussed in the 'Query Phase' sub-section.

A. *TRAINING PHASE*

In the training phase, the domain knowledge of a CDO and its constraints are mapped into the Q table. This takes place in three steps: 1) the CDO and its constraints are encoded numerically to enable mapping to the reward table (R-table); 2) the R-table is initialized followed by the Q-table update by iterating through the Bellman's equation; 3) the R-table is updated.

1) NUMERICAL ENCODING OF CDO

To enable processing of a CDO with Q-learning, all the decision variables must be encoded numerically. For example, the Ball CDO's events can be represented as small=0, medium=1, large=2, white=3, brown=4, orange=5, golf=6, football=7, and basketball=8 (see Fig. 2). Based on the constraints for the given CDO, a set of tuples will be constructed to represent the constraints, as shown in Table 4.

Each tuple will be arranged in ascending order based on its event numbers. Rearranging the constraints does not violate the constraint's definition (due to the integrity of constraints). Each tuple is considered as a complete path where the first number is the starting point, and the last number is the goal point. During the training phase, all the constraints are mapped within the Q-table by strengthening the relationship among events of each constraint.

The total number of CDO events defines the size of the R-table and the Q-table. For the Ball CDO (see Fig. 2), both the R-table and Q-table will be structured as 9 × 9 matrices as there are 9 events (small, medium, large, white, brown, orange, golf, football, and basketball). The R-table and Q-table will be initialized with -1 and 0, respectively. The R-table will be updated based on the tuples, while the Q-table will be optimized using the Q-learning algorithm (see Algorithm 1). There are three phases in the training process: 1) the R-table is first preprogrammed and used to generate the Q-table, 2) the Q-table is updated using Algorithm 1, and 3) the R-table is updated once more to correct the search directions.

2) FIRST UPDATE OF THE REWARD TABLE AND Q-TABLE

The tuples that are derived from the constraints will be mapped within the R-table for the immediate reward calculation. A consecutive set of ascending ordered number pairs, {(x1, y1),(x2,y2),...,(xi,yi),...}, will be generated from the tuples. Each (xi,yi) represents a position in the R-table. Only consecutive adjacent number pairs are considered in this step, while other pairs are considered in step 3. For example, if the generated tuple is (0,1,4,5,7,9), then in this step the considered number pairs are: {(0,1),(1,4),(4,5),(5,7),(7,9)}.

Thus, from the tuple (0, 3, 6) in Table 4, two ascending ordered number pairs, (0, 3) and (3,6), will be generated. The number pair that contains the goal point (last number in tuple) in the y-coordinate position, i.e. (3,6), will be marked as 100 in the R-table. The opposite coordinate of that pair, i.e. (6,3), will be marked as 0 in the R-table. The remaining number pairs (such as (0,3)) along with their opposite pairs (such as (3,0)) will be marked as 0 in the R-table. Table 5 shows the generated number pairs from the constraints of Table 4 along their values in the R-table. There are three 100 values and nine 0 values in Table 5, as there are three constraints considered. All other positions in the Reward table will be initialized to -1.

There may be some situations where multiple constraints share the same events. Let us consider a scenario where one more choice point called 'Field Type' is added to the ball CDO (Fig. 2). 'Field Type' will have two events: 'indoor' and 'outdoor' that will be represented by numerical values 9 and 10, respectively. The updated constraints, along with their tuple representation, are shown in Table 6.

These sharing events' positions can be anywhere in the tuples based on each event's unique numbering strategy,

TABLE 5. Reward table's values for constraints in table 4.

Number Representation of Constraints	Generated Number Pair with Values in Reward Table
(0, 3, 6)	(0,3) = 0; (3,0) = 0; (3,6) = 100; (6,3) = 0
(1, 4, 7)	(1,4) = 0; (4,1) = 0; (4,7) = 100; (7,4) = 0
(2, 5, 8)	(2,5) = 0; (5,2) = 0; (5,8) = 100; (8,5) = 0

TABLE 6. Updated tuple representations of constraints.

Constraints	Number Representation
Small, White, Golf, Outdoor	(0, 3, 6, 10)
Medium, Brown, Football, Outdoor	(1, 4, 7, 10)
Large, Orange, Basketball, Indoor	(2, 5, 8, 9)

as the tuples are arranged in ascending order. If the 'indoor' and 'outdoor' events of the CDO are represented with numerical values 3 and 4 respectively, then the tuples will be rearranged as '*Small, Outdoor, White, Golf*' => (0, 3, 5, 8), '*Medium, Outdoor, Brown, Football*' => (1, 3, 6, 9), and '*Large, Indoor, Orange, Basketball*' => (2, 4, 7, 10).

A new set of number pairs will be generated from the constraints of Table 6 for the R-table. An example of generated number pairs for the first constraint ('*Small, White, Golf, Outdoor*' => 0, 3, 6, 10) from Table 6 will be (0,3) = 0; (3,0) = 0; (3,6) = 0; (6,3) = 0; (6,10) =100; (10,6) = 0. The immediate reward value of the Bellman's equation will be calculated from the R-table. Afterwards, the Q-table will be trained using Algorithm 1.

3) SECOND UPDATE OF THE REWARD TABLE

After training the Q-table, the R-table will be updated once again using the same tuples. Some new information will be incorporated into the R-table without changing any previously added values.

As with the prior R-table update process, a list of ascending ordered number pairs will be generated from each tuple. Those new number pairs were ignored in the first R-table update process. Thus, for the tuple (0,1,4,5,7,9), the new number pairs considered are: {(0,4), (0,5), (0,7), (0,9), (1,5), (1,7), (1,9), (4,7), (4,9), (5,9)}.

For the Ball CDO example, the tuple (0, 3, 6) in Table 4 will generate number pair (0,6). Those newly generated number pairs (positions in the R-table) will be marked with any other number except for 0 and 100 in the R-table. During the solution generation process, the algorithm looks for the values in the R-table to continue the search in a particular direction. The algorithm stops the search if it finds out that the R-table contains '-1' in that search direction. In this experiment, those new number pairs are updated with a value of 10 in the R-table. An example of the updated R-table (used during the query phase) for the ball CDO (Fig. 2) is shown in Table 7.

TABLE 7. Updated reward table for ball CDO of fig. 2.

Action / State	0	1	2	3	4	5	6	7	8
0	-1	-1	-1	0	-1	-1	10	-1	-1
1	-1	-1	-1	-1	0	-1	-1	10	-1
2	-1	-1	-1	-1	-1	0	-1	-1	10
3	0	-1	-1	-1	-1	-1	100	-1	-1
4	-1	0	-1	-1	-1	-1	-1	100	-1
5	-1	-1	0	-1	-1	-1	-1	-1	100
6	-1	-1	-1	0	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	0	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	0	-1	-1	-1

B. QUERY PHASE (EXTRACTING DATA FROM Q-TABLE)

According to the definition of a CDO, a user can assert any set of events as input, and the solver will generate valid solutions based on the assertions and constraints. In optimal pathfinding problems, start and endpoints are required to find the optimal path from the Q-table. On the contrary, there is no concept of starting and ending information in the CDO. A novel knowledge extraction algorithm (Algorithm 2) is developed to extract all the solutions from the Q-table based on either a single assertion or a set of assertions. Unlike Q-learning, this algorithm does not use the maximum function to extract information. Based on the assertions (starting point), the algorithm traverses the Q-table in all directions to obtain valid solutions. The algorithm uses the final updated R-table to validate the consistency of the solutions with the constraints.

Algorithm 2 describes the knowledge extraction pseudocode. The algorithm uses the trained Q-table and R-table to generate solutions. Table 8 shows the trained and normalized Q-table of the ball CDO (Fig. 2) that was generated during the training phase. Here in the Q-table, each ‘state’ and ‘action’ represents a particular event of the CDO, i.e. state = 0 and action = 0 represent the event, ‘small’ of the CDO. Other events ‘state & action’ representations are: small (state & action =0), medium (state & action =1), large (state & action =2), white (state & action =3), brown (state & action =4), orange (state & action =5), golf (state & action =6), football (state & action =7), and basketball (state & action =8).

During the solution generation process, ‘state’ represents the current event of the CDO, which is already the part of the solution, and ‘action’ presents the future events of the solution, which will be the part of the solution.

A user can also assert one or multiple events as input. If an event is asserted as an input state, then the algorithm selects that event as the starting state. Otherwise, the algorithm selects the starting state from the input states randomly (step 1 of Algorithm 2). Asserting more input states helps the user find narrower sets of solutions. In that scenario, all

Algorithm 2 Knowledge Extraction Phase Pseudocode

```
INPUT: input_states (in_s), Q-table, and R-table
OUTPUT: {solutions}
INITIALIZATION:
  1. Select (start_state) from (in_s) randomly.
  2. Assign {validation_list}.
  3. Load Q-table, and R-table.
STEPS:
  4. Select ∀(next_state): Q-table[start_state,] > 0
  5. Perform search.
   i IF ∀(next_state) > (start_state) THEN
       FOR each_state in ∀(next_state):
         IF R-table [start_state, each_state] != −1 THEN
           -Generate {solutions} using Forward_search.
   ii ELSE IF (next_state) < (start_state) THEN
       FOR each_state in (next_state):
         IF R-table [each_state, start_state] != −1 THEN
           -Generate {solutions} using Backward_search.
   iii ELSE
     Select ∀ (backward_states), and ∀ (forward_states)
     ∀ (backward_states): Q-table[start_state,] < start_state
     ∀ (forward_states): Q-table[start_state,] > start_state
     FOR each_state in ∀ (backward_states):
       IF R-table [each_state, start_state] != −1
         -Generate partial_solution,
         FOR each_state in ∀ (forward_states):
           IF R-table [start_state, each_state] != −1
             -Generate solution from partial_solution,
  6. IF len(validation_list) != 0 THEN,
       FOR each_solution in {set of solution}
         IF {validation_list} ∉ each_solution THEN
           Remove each_solution from {set of solution}
```

TABLE 8. Trained and normalized q-table for ball CDO of fig. 1.

Action / State	0 Small	1 Med	2 Large	3 White	4 Brown	5 Orange	6 Golf	7 Football	8 Basketball
0: Small	0	0	0	80:F1	0	0	0	0	0
1: Medium	0	0	0	0	80	0	0	0	0
2: Large	0	0	0	0	0	80	0	0	0
3: White	64	0	0	0	0	0	100:F2	0	0
4: Brown	0	64:B2	0	0	0	0	0	100	0
5: Orange	0	0	64:M1	0	0	0	0	0	100:M1
6: Golf	0	0	0	80	0	0	0	0	0
7: Football	0	0	0	0	80:B1	0	0	0	0
8: Basketball	0	0	0	0	0	80	0	0	0

other states except for the starting state are assigned to the validation list (step 2). The validation list is used to check the validity of solutions based on assertions. The algorithm starts its search based on the starting state (event) and removes those solutions which do not contain events from the validation list as each solution must contain all the input states (events) provided by the user.

Based on the starting state and its corresponding next states' value, the algorithm selects one of the three search strategies to generate solutions. These three search strategies are forward search, backward search, or bi-directional search (which consists of both forward and backward searches). One of these three search options are selected in step 5 under options i, ii, or iii. The details of these search processes are described below with examples.

1) FORWARD SEARCH

In every step, the algorithm determines all the available states that it can traverse to form the current state. All the values in the trained Q-table's current state row will be zero other than the traversable states. All those traversable states are considered as possible next states. The selected next states' numerical values will either be greater or less than the starting state's numerical value as the events of the CDO are represented with unique numbers. If all the next states' numerical values are greater than the starting state's numerical value, then the algorithm traverses in the forward direction only. This forward directional search is described as $\text{Forward}_{\text{search}}$ in the algorithm (step 5i). In this forward search approach, the algorithm will only consider those next states whose numerical values are higher than the current state. In this manner, the algorithm explores new states and appends them to the starting state. If more than one next state is available, then the algorithm branches the search and generates separate solutions for each branch. The algorithm will stop the search when there are no higher numerical states than the current state.

Let's consider a user asserted 'small' as an input state, then the starting state will be 0 in this scenario as the numerical value of 'small' is 0 ('states' from Table 8). Also, as only event is asserted as input, the validation list will be empty here. As the next traversable state is 3 for the starting state 0 (only Q [0,3] has a value above zero which is marked as 'F1' in Table 8), the algorithm will only perform $\text{Forward}_{\text{search}}$ in that scenario. From the state 3, the algorithm will only consider 6 (Q[3,6]: F2 in Table 8) as the next state, as 6 is greater than 3. Afterward, the algorithm will look for the next states in the Q[6,_] row. As there is no next state available (columns labelled 7 and 8 do not have values above zero) the algorithm will stop its search and return (0,3,6) as its solution. Here, (0, 3, 6) stands for (small, white, golf).

2) BACKWARD SEARCH

If all the numerical values of the next states are smaller than the starting state, then the algorithm will traverse in the backward direction only. This backward directional search is described as $\text{Backward}_{\text{search}}$ in Algorithm 2 (step 5ii). In this search process, the algorithm only considers those next states, whose numerical values are smaller than the current state.

For example, if 'football' is asserted as the input state, then the next state will be 4 (Q[7,4]: B1 in Table 8). Here 'football's state value is 7, and only Q[7,4] has values above zero. As 4 is smaller than 7, the algorithm will only perform the $\text{Backward}_{\text{search}}$ for this assertion. From state 4, the algorithm will pick 1 as next state as Q[4,1] (B2 in Table 8) has values above zero, and the numerical value 1 is less than 4. For the assertion of 'football', the algorithm will return (7, 4, 1) as its solution.

The R-table helps the algorithm decide which future states to consider maintaining the constraints' consistency. The algorithm uses (3) for $\text{Forward}_{\text{search}}$ (step 5i) and (4) for $\text{Backward}_{\text{search}}$ (step 5ii) to decide the validity of the next state. This feature of the algorithm is useful when the constraints share events among themselves.

$$R\,[start\ state, next\ state]! = -1 \tag{3}$$

$$R\,[next\ state, start\ state]! = -1 \tag{4}$$

3) BI-DIRECTIONAL SEARCH

There could be a scenario where some of the next states' numerical values are higher than the starting state and some are not. In that scenario, the algorithm has to travel in both directions to generate a complete solution (step 5iii). Let's consider a user asserted 'orange' as an input state (state = 5). For this assertion, the algorithm will consider both state 2 and 8 (M1 in Table 8) as Q[5,2] and Q [5], [8] have values above zero. First, the algorithm will perform $\text{Backward}_{\text{search}}$ considering 'state = 2' as the next state and will generate a partial solution. In the $\text{Backward}_{\text{search}}$, the algorithm will only consider those next states, whose numerical values are smaller than the current state. After that, the algorithm will perform $\text{Forward}_{\text{search}}$ considering 'state = 8' as the next state. The algorithm will append newly found states with the previously generated partial solutions. As mentioned earlier, the $\text{Forward}_{\text{search}}$ algorithm will only consider those next states whose numerical values are higher than the current state. After traveling both directions, the algorithm will return a complete solution to the user.

Eventually, If the validation list is not empty, the algorithm removes those solutions that don't contain events from the validation list (step 6).

C. HIGHER ORDERED INSTANCE SEARCH

The instance node replicates the substructure of a CDO. As described in section III-B, each substructure of the CDO holds partial solutions. For higher-ordered instances, all of an instance's information, along with its constraints, are mapped within the same Q-table. For the ball CDO in Fig. 2, 'n' separate CDO instances are considered for mapping into the Q-table. Also, for the higher-ordered instance setting, all the events of each instance will be encoded with unique numbers.

Algorithm 3 describes the knowledge extraction pseudocode for the multi-instance scenario. Algorithm 3 uses Algorithm 2 to generate partial solutions for each instance and finally concatenates them according to the problem definition. Algorithm 3 first separates the inputs based on instances and determines the list of asserted and unasserted

Algorithm 3 Multi Instance Knowledge Extraction

```
INPUT: input_states (in_s), Q-table, and R-table
OUTPUT: {solutions}
INITIALIZATION:
1. Sort inputs based on Instances.
2. Find asserted and unasserted instances.
3. FOR each_instance in ∀ (Instances):
4.      IF each_instance ϵ ∀(asserted instance):
5.         – Generate partial solutions using Algorithm 2
6.      Else IF each_instance ϵ ∀(unasserted instance):
7.         – Generate partial solutions using Algorithm 2
             (mostly some of the partial solution)
8.      Concatenate partial solutions.
```

instances. A user can assert input events for all instances, or some of the instances. If the 'n' in Fig. 2 is set to 3, then there will be three replicas of the ball CDO. If the user asserts 'small' for the first instance and 'large' for the second instance, then the instances 'one' and 'two' will be members of the asserted instances list, and 'three' will be a member of the unasserted list. Algorithm 3 generates all the partial solutions for the asserted instances. For the unasserted instances, algorithm 3 generates some of the partial solutions based on users' interests.

The Q-learning-based CDO solving approach's computational complexity depends on the number of solutions to a problem. The number of solutions depends on the problem definition, number of instances, and the user's assertions. The Q-learning-based approach has quadratic time ($O(n^2)$) complexity.

Two flags are used to control the total number of generated solutions. These flags are 'Generate only one solution,' and 'Generate all possible instance-level combinations for the asserted values.' Three different scenarios are experimented using a combination of two flags. For higher-order instances, the total number of solutions increases exponentially. As there is no objective function to find the best solution, these flags can be used to control the total number of generated solutions.

There are a handful of solvers that have been used to solve CDOs including Sherlock (based on the Choco 2.15 constraint solver library [26]), WAKA solver (a Java based solver developed by AFRL), and a path-based forward checking algorithm [11], [25]. There is both a serial [25] and a parallel [11] version of the path-based algorithm. These existing solvers can generate either only one or all the solutions. However, most of these solvers cannot handle the higher-order instance property of CDOs quickly, typically taking multiple weeks to solve the problems evaluated in this study. Only the parallel version of the path based forward checking algorithm can handle higher-order CDO instances and generate solutions in runtime. These solvers will require large amounts of memory and power to deal with larger CDOs as there is no control to limit the total number of solutions generated. Thus, it is quite difficult for an agent to generate solutions for higher-order instances in low power consuming platforms using these previous algorithms. The two flags introduced to handle this scenario will enable the solver to generate different numbers of solutions based on user requirements.

When the 'Generate only one solution' flag is turned on, Algorithm 3 will generate only one complete solution for the problem. Usually, each instance generates a vast amount of partial solutions based on assertions and constraints. As described before in section III-B, the algorithm creates complete solutions by concatenating partial solutions; it will randomly generate only one partial solution from each instance.

When the 'Generate all possible instance-level combinations for the asserted values' flag is turned on, Algorithm 3 generates all possible combinations of the partial solutions for each asserted instance based on the asserted inputs. Instead of generating all partial solutions for the unasserted instances, the algorithm generates only one partial solution for this flag. Algorithm 3 then finally concatenates those partial solutions and makes several complete solutions for that problem. From the previous example, Algorithm 3 will generate all the partial solutions for the 'first' and 'second' instances but will generate only one partial solution from the 'third' instance.

When both flags are turned off, the algorithm generates all possible combinations of the partial solutions for the asserted instances based on assertions. For unasserted instances, the algorithm randomly selects an event from the unasserted instance and generates all possible combinations of partial solutions for that instance.

A solution threshold is used to prevent the algorithm from being out of memory. This option is enabled only for the higher-ordered instance settings and low memory devices. After this threshold, the algorithm will stop adding new solutions to the existing solution list. This feature will prevent a low power device from facing any memory related issues during run time.

VII. EXPERIMENTAL SETUP

A. EXPERIMENTAL CDO

The broadcast CDO shown in Fig. 5 was used to evaluate the performance of the Q-learning based algorithm in this study. This CDO helps users figure out the relation among broadcasting companies and their associated stations and channels.

This CDO has three different broadcasting companies (abc, cbs, and nbc), three different stations (kabc, kcbs, and knbc), and five different channels (ch1 to ch5). These elements are the 'Events' of the broadcast CDO. In this Broadcast CDO, three different 'Instance' nodes are used, which are Networks, StationMul, and ChannelMul. This broadcast CDO contains all the entities of a basic CDO, like 'Event,' 'Instance,' 'ChoicePoints,' and 'SubParts.' Six constraints were used in this study, as shown in Table 9. The six constraints were used on all levels of instances.

Table 10 shows the solution search space for the broadcast CDO with and without constraints for different instance

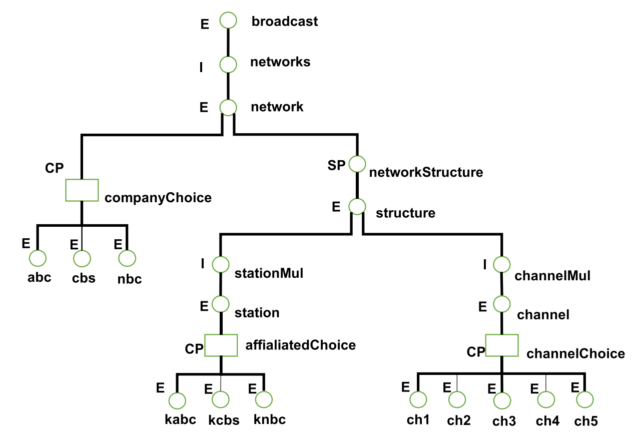


FIGURE 5. **Domain structure of broadcast CDO.**

TABLE 9. **User-defined constraints for the broadcast CDO.**

Specification (User Defined Constraints)
Iff company is cbs, then channel is ch4.
Iff company is cbs, then station is kcbs.
Iff company is abc, then channel is ch1 or ch2.
Iff company is abc, then station is kabc.
Iff company is nbc, then channel is ch3 or ch5.
Iff company is nbc, then station is knbc.

TABLE 10. **Broadcast cdo's solution space.**

Instances	Search Space without Constraints	Search Space with Constraints
1	45	5
2	455,625	81
3	1.04×10^{12}	4913
4	5.32×10^{20}	1,185,921
7	9.23×10^{60}	7.41×10^{16}
10	2.40×10^{122}	1.30×10^{33}

settings. An instance value of 4 means that all instance nodes: Networks, StationMul, and ChannelMul are set to 4. The same configuration goes for other instance numbers. The solution search space without constraints is calculated as $[(3^i)*(3^i*5^i)^i]$ where i is the instance number. As the instance number increases, the size of the solution space increases exponentially. Constraints reduce the search space of the problem.

B. HARDWARE

Two low power devices were used to map and mine knowledge in this paper: the Asus Tinker Board and the Intel NUC. The Tinker Board is a single-board computer that has a Rockchip Quad-Core RK3288 processor (Quad-core ARM Cortex-A17 @upto 1.8GHZ) and 2GB DDR3 RAM. The maximum power consumption is 5W. The

TABLE 11. **Three simulation configuration.**

Flags Representation	Scenarios
FF	Both flags are false
TF	Only 'Generate only one solution' flag is on
FT	Only 'Generate all possible instance-level combination for the asserted values' flag is on

TABLE 12. **Assertions used in the experiments.**

Instance	Assertion
2	abc{1}, ch1{1_1}
7	abc{1}, ch1{1_1}, ch1{1_2}, cbs{2}, cbs{3}, cbs{4}, cbs{7}
10	abc{1}, ch1{1_1}, ch1{1_2}, ch2{1_3}, ch2{1_4}, ch2{1_7}, cbs{2}, cbs{3}, cbs{4}, cbs{5}, cbs{6}, cbs{7}, cbs{8}, nbc{9}, ch3{9_5}, ch5{9_6}, ch3{9_8}, ch3{9_9}

Intel NUC (NUC8i7BEH) had an Intel Core i7-8559U processor (2.7 GHz - 4.5 GHz, Quad-Core, 8MB Cache) and 16GB of DDR4-2400 RAM. The NUC's idle state power ranged from 3W to 7W [37]. The Tinker Board and NUC utilized Debian and Ubuntu 16.04 as their operating system, respectively.

C. EXPERIMENTAL CONFIGURATION

The algorithm was developed using Python. To verify the scalability of the solver, the Broadcast CDO was tested with three different instance settings (2, 7, and 10). As shown in Table 10, the solution search space for these instance settings is enormous. The combination of two flags ('Generate only one solution' and 'Generate all possible instance-level combinations for the asserted values') made it possible to use three different scenarios in this study to understand the performance of the solver, as shown in Table 11.

The algorithm will generate different numbers of solutions depending on flag setting. In this experiment, for 10 instances, the solution threshold was set to 500,000 on the Asus Tinker Board. As the solution space is enormous for 10 instances, and the Tinker Board has less memory (2GB), this solution threshold is enabled for the Tinker Board (but not the NUC).

The assertions (inputs) used in this study for three instance settings are shown in Table 12. Assertions ['abc_1', 'ch1_1_1'] for 2 instances (see Table 12) means that the partial solutions from the first instance must contain information about 'abc' and 'ch1'. The second instance's partial solutions can be any valid combination from the broadcast CDO. The examples of two complete solutions for 2 instances are: [*abc{1}, kabc{1_1}, kabc{1_2}, ch1{1_1}, ch1{1_2}, cbs{2}, kcbs{2_1}, kcbs{2_2}, ch4{2_1}, ch4{2_2}*] and [*abc{1}, kabc{1_1}, kabc{1_2}, ch1{1_1}, ch1{1_2}, nbc{2}, knbc{2_1}, knbc{2_2}, ch3{2_1}, ch3{2_2}*].

TABLE 13. Average training time on both devices.

Instances	Iterations	Tinker Board Training Time (Seconds)	NUC Training Time (Seconds)
2	5000	3.274	0.334
7	20000	137.38	14.143
10	50000	1273.49	251.67

As the algorithm selects an event randomly for the unasserted instances for the 'both flags turned off scenario,' there is a randomness in the number of generated complete solutions. The number of generated solutions can be predetermined by fixing the assertions for all instance levels.

VIII. RESULT AND DISCUSSION

A. RESULT GENERATION

We validated the algorithm by comparing its outputs for the CDOs listed earlier in the paper against known correct results for a variety of assertions and constraints. The solutions generated for smaller sized CDOs (such as the Ball CDO) were hand-verified. For larger CDOs (such as the Broadcast CDO), solutions were compared to those generated by the previously verified parallel CDO solver in [11]. That solver utilized a novel parallel path-based forward checking algorithm to solve CDOs at high speeds and was verified against a variety of other solvers (all slower), including Screamer+ [23] and Sherlock [26] During validation, only the 'FF' flag setting was considered as all the existing solvers can either generate only one or all the solutions. With the 'FF' setting, the Q learning algorithm generates all possible solution combinations for the asserted instances.

B. TIMING

The Q-learning approach updates the Q-table by iterating through the Algorithm 1 during the training phase. This training process requires just a CDO definition along with the constraints. As the training algorithm is a serial process, the training time depends on the number of iterations required, which in turn depends on the size and complexity of the CDO. The Q-table can be trained in low power devices in case online adaptation is ever required. Table 13 shows the average training time required for all the instances on both low power devices. For a specific CDO and set of constraints, the R-table and Q-table need to be trained only once.

As described in Section III-B, in multi-instance settings, the algorithm generates each instance's solutions (partial solutions) separately and then concatenates them. Generating all the partial solutions for each instance is very fast on both the Tinker Board and NUC. Table 14 shows the average time required to create all the partial solutions.

For example, the algorithm will generate ten sets of partial solutions for 10 instances of the Broadcast CDO. For generating each of these ten sets of partial solutions, the algorithm required an average of 62ms and 4.9ms on the Tinker Board and NUC, respectively. The total number of partial solutions for each instance depends on the assertions and constraints. An example of multiple numbers of partial solutions is shown later in this paper in Table 19.

TABLE 14. Average time to generate partial solutions.

Instances	Total Set of Partial Solution	Tinker Board Partial solution generating time (ms)	NUC Partial solution generating time (ms)
2	2	2	0.2
7	7	9.5	0.986
10	10	62	4.9

TABLE 15. Time to generate a complete solution for flag 'TF'.

Instances	Solution	Tinker Board Time (ms) (Loading+Generate)	NUC Time (ms) (Loading+Generate)
2	1	66.03 (62.6+3.442)	19.8 (19.2+0.62)
7	1	544.54 (465.0+79.5)	84.4 (75.8+8.6)
10	1	12813.2 (12256.0+557.2)	1909.02 (1812.6+96.416)

TABLE 16. Time to generate complete solutions for flag 'FT'.

Instances	Solution	Tinker Board Time (ms) (Loading+Generate)	NUC Time (ms) (Loading+Generate)
2	2	64.11 (61.3+2.86)	19.6 (19.0+0.58)
7	32	547.3 (474.7+72.63)	84.81 (76.3+8.535)
10	2048	12878.6 (12259.0+619.98)	1877.63 (1784.6+93.01)

The number of solutions generated will depend on the flags selected from Table 11. For the 'TF' flag, the algorithm will only generate one complete solution. Table 15 shows the average time required to generate one complete solution in milliseconds for three different instances.

With the 'FT' flag, the algorithm will generate a fixed number of solutions based on assertions for different instances. Table 16 shows the average time needed to generate 2, 32, and 2048 complete solutions for 2, 7, and 10 instances respectively with the 'FT' flag. The actual solution generation process is faster than the table loading time for both 'TF' and 'FT' flag settings. The required time to load the tables and generate complete solutions (by concatenating partial solutions from all instances) increases with the increase in instances.

TABLE 17. Time to generate complete solutions for flag 'FF'.

		Tinker Board	NUC
Instances	Solution	Time (ms) (Loading+Generate)	Time (ms) (Loading+Generate)
2	2 or 8	64.3 (61.5+2.75)	20.1 (19.5+0.601)
7	524,288	13,413.00 (467.7+12,945.7)	2566.7 (75.6+2491.0)
	4096	581.6 (474.9+106.7)	90.3 (76.57+18.25)
	32	507.43 (467.4+39.9)	81.0 (76.5+4.1)
10	2,097,152	2048*	12,782.8 (1779.6+11,003.2)
	2048	13,006.8 (12,488.5+518.3)	1873.0 (1788.4+84.18)

* Algorithm generates 2048 complete solutions due to threshold flag

As mentioned in section VI-C, with the 'FF' flag, the algorithm selects the starting state randomly for the unasserted instances. This can lead to a variation in the number of solutions generated depending on which assertions are selected by the algorithm. Table 17 shows the possible number solutions and time needed for the 2, 7, and 10 instance cases with the 'FF' flag. In the 2 instance case, it generates either 2 or 8 complete solutions for the CDO. If the algorithm selects channel '*cbs*' for the unasserted instance, then 2 complete solutions will be generated. If either '*abc*' or '*nbc*' are selected, then there will be 8 complete solutions. The example of two complete solutions for 2 instance scenarios were shown in section VII-C.

Most of the runtime is utilized in concatenating partial solutions to generate complete solutions rather than creating partial solutions. To generate all ten sets of partial solutions for 10 instances, the algorithm spends around 50ms (4.9ms×10, see Table 14), whereas it spends around 10,953ms (11003.2 - 50) and 38.18ms (84.18 - 50) to concatenate those partial solutions for generating 2,097,152 and 2048 complete solutions respectively in the NUC.

For the higher-order instances, the algorithm needs to concatenate a large number of partial solutions to create complete solutions. In low power devices, the onboard memory is often not enough to store all the solutions, causing the algorithm to stop when it runs out of memory. The average time required to reach a solution is slightly higher in the Tinker Board compared to the NUC because of the low-speed processor used in the Tinker Board. The NUC is around 6 to 7 times faster than the Tinker Board for higher-order problems.

As shown in Table 17, the maximum number of solutions generated for instances 2, 7, and 10 are 8, 524,288, and 2,097,152. In case of 10 instances, the maximum number of solutions cannot be handled by the limited memory of the Tinker Board (2GB), and thus the threshold flag needs to be used for this case. Due to this threshold flag, the algorithm checks the total number of future complete solutions after executing each instance and triggers the 'generate only one solution' flag when the total number of future complete solutions reaches the threshold value, e.g., 500,000. Thus, after concatenation, the total number of complete solutions will always remain 2048 in the Tinker Board, for 10 instances in this study.

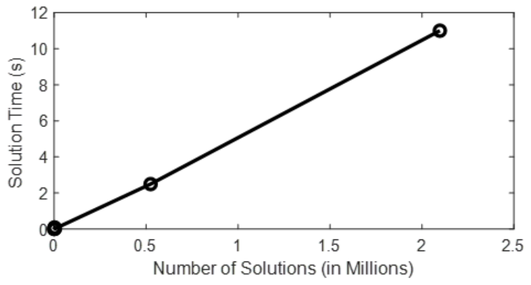


FIGURE 6. Gradual solving time increment with the number of solutions.

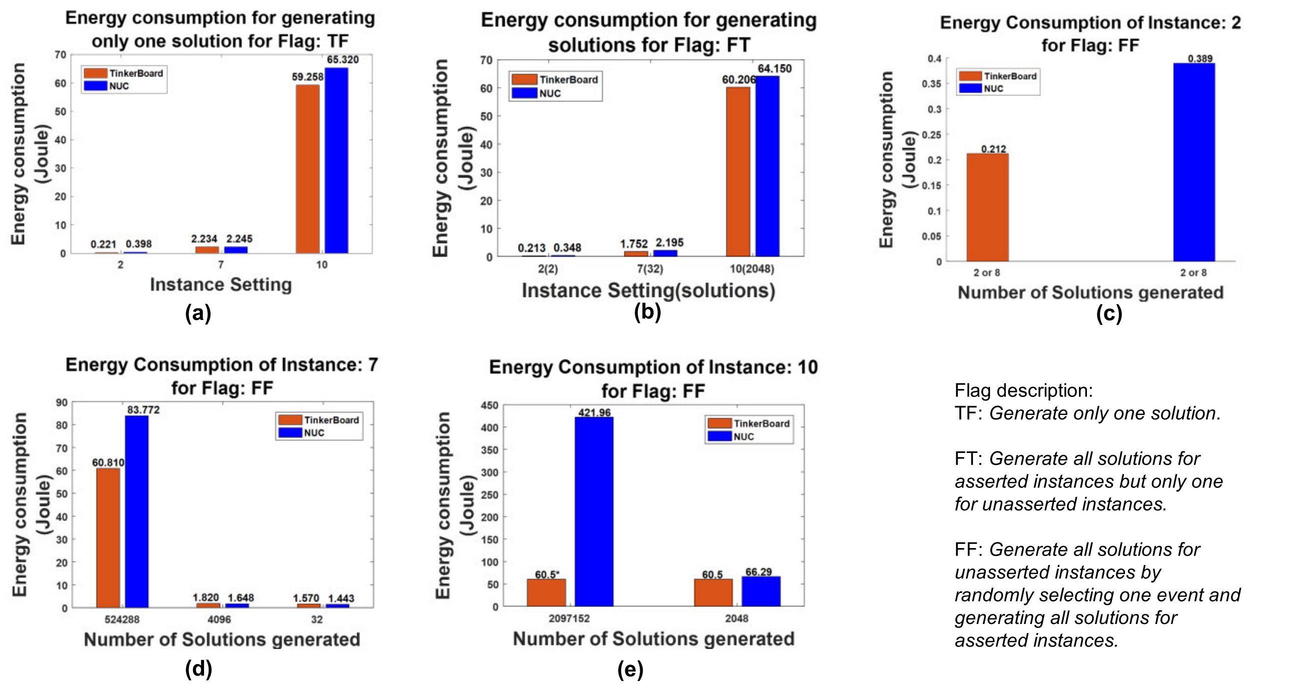
To show the scalability of the Q-learning-based approach on real-world problems, different-sized problems are considered as the solving time depends on the number of solutions. In the Broadcast CDO experiment, the number of solutions varied from only 8 to 2,097,152 solutions using different instance settings. Fig. 6 shows the gradual solving time increment with the number of solutions on hardware NUC. In Fig. 6, only the solution generation time (seconds) is considered, not the Q-table loading time. The Q-learning-based approach can easily handle real-world problems as it generated 2,097,152 complete solutions in 11.2 seconds.

C. ENERGY CALCULATION

A key benefit of this approach is its ability to run on low power processing systems. To evaluate this, the energy consumed by the algorithm was measured on the NUC and the Tinker Board using a 'Watts Up Pro' power meter. The meter was configured to collect power consumption every second. In both devices, the algorithm was executed multiple times consecutively to collect stable power consumption data. The idle state power of the system was not deducted from the actual power consumption. On the NUC, the idle state power was about 5.3W, while the active power ranged from 20W to 35W. On the Tinker Board, the idle power was about 3W, while the active power ranged from 3.3W to 4.5W.

A comparison of energy consumptions for the flag 'TF' and 'FT' on the Tinker Board and NUC are shown in Fig. 7(a) and 7(b), respectively. The algorithm generates only one complete solution for the flag 'TF' (see Fig 7(a)) and generates a fixed number of complete solutions based on the assertions and instances for the flag 'FT' (see Fig 7(b)). As the number of solutions increases with the instance numbers, the required energy to generate these solutions also increases.

With the 'FF' flag, the number of solutions generated depends on the assertions. Figs. 7(c), 7(d), and 7(e) show the energy consumption for all three instances (2, 7, 10) due to the flag 'FF' while generating all possible solutions,



Flag description:
TF: *Generate only one solution.*

FT: *Generate all solutions for asserted instances but only one for unasserted instances.*

FF: *Generate all solutions for unasserted instances by randomly selecting one event and generating all solutions for asserted instances.*

FIGURE 7. The instance settings and the number of partial solutions determine the amount of energy the algorithm will consume. Energy consumption for (a) all three instances with flag 'TF', (b) all three instances due to flag 'FT', (c) 2 instances when using flag 'FF', (d) 7 instances for flag 'FF', and (e) 10 instances with the flag 'FF'.

respectively (except for instance 10 on the Tinker Board due to memory limitations). As a complete solution is generated by concatenating partial solutions, the energy consumption increases when there are more partial solutions.

For 2 instances, the energy consumption is almost the same for all the flag settings (TF, FT, FF) because the solution search space is small. For 7 instances, the algorithm consumes around 61J and 83J on the Tinker Board and NUC, respectively, to generate 524,288 complete solutions. In contrast, the energy consumption is less than 2J for generating 4096 or 32 complete solutions on both devices for 7 instances. For 10 instances, we use the threshold flag on the Tinker Board to generate only 2048 complete solutions (instead of 2,097,152 with flag 'FF'). Thus for 10 instances, the NUC consumes 421J to generate 2,097,152 complete solutions, whereas both the Tinker Board and the NUC consume around 65J to generate 2048 complete solutions.

D. PERFORMANCE COMPARISON

There are only a few CDO solvers available, namely Sherlock (based on the Choco 2.15 constraint solver library [26]), WAKA solver (Java-based solver developed by AFRL), an exhaustive depth-first search based algorithm [7], and both serial and parallel versions of the path-based forward checking algorithm [11] [25]. Except for the parallel path-based forward checking algorithm [11], none of the existing CDO solving approaches can solve large and complex CDOs in runtime. This GPU enabled parallel path-based forward checking algorithm solves large CDOs on high-performance computing systems, which can utilize multi-node systems by using MPI. The parallel path-based forward checking algorithm [11] is superior in performance compared to other existing CDO solvers. Therefore, we compare the Q-learning based algorithm with the algorithm in [11], as this is the fastest algorithm, we are aware of for solving CDOs.

The path-based forward checking algorithm generates all the possible partial solutions that can be generated based on the assertions and constraints. On the other hand, as the Q-learning based approach is designed to execute on low power consuming devices, it does not generate all the possible partial solutions.

The Q-learning based algorithm generates solutions based on the flags, assertions, and constraints. To compare the performance of the solvers, the Q-learning based solver is modified by incorporating for loops to generate an equal number of solutions generated by the path-based solver.

In this performance comparison, the Broadcast CDO with 12 instances is considered, as the solution space is huge and both approaches need to use all their resources. The assertions shown in Table 18 are used in this case, while using the same constraints shown in Table 9. A system with an Intel Xeon E5-1620v2 3.70GHz processor, 64 GB of DRAM, and an NVIDIA Tesla K80 was used to run the path-based algorithm. The NVIDIA Tesla K80 has 4992 CUDA cores.

Nine loops are used in the Q-learning based approach to match the exact number of partial solutions generated by the path-based search. In this study, only partial solutions are generated instead of complete solutions (which the

TABLE 18. **Assertions used for performance comparison.**

Instances	Assertion
12	abc{1}, abc{2}, nbc{4}, nbc{5}, nbc{6}, cbs{8}, cbs{9}, cbs{10}, abc{11}, nbc{12}

path-based search normally does). As both the algorithms are not generating complete solutions by concatenating the partial solutions, there is no out of memory scenario for the Tinker Board running the Q-learning algorithm.

The time complexity of the Q-learning-based CDO solving approach and GPU enable parallel path-based forward checking algorithm depends on the number of solutions in a problem. Both algorithms face quadratic time complexity (O(n2)) with the increase in the instance property for the Broadcast CDO. The advantage of this Q-learning-based approach is that this approach can solve large and complex CDOs in runtime on low power platforms, which none of the existing CDO solvers can do.

Table 19 shows all the nine sets of generated solution structures, for the Broadcast CDO with 12 instances. Both the path-based and Q learning based approach generated the same amount of partial solutions. In Table 23, each row represents one of nine generated solution sets, and each solution set consists of twelve sets of partial solutions. As we only have three company choice events: ‘abc,’ ’nbc,’ and ‘cbs’, there will be repetition in the partial solution selection process. In table 19, each event is followed by the number of partial solutions generated within parenthesis. For example, the solution set “abc(4096), abc(4096), cbs(1)…” indicates that the algorithm produced 4096 partial solutions related to ’*abc*’ for the first and second instances and 1 partial solution related to ’*cbs*’ for the third instance. This continues on for the remaining nine instances. All of these partial solutions constitute the complete solution for the set.

Table 20 shows the required time and energy to generate all the partial solutions for both the path-based and Q-learning based approaches. The path-based search consumes on average 210W for generating all the partial solutions based on the assertions. On the other hand, the Q-learning based approach consumes around 29W and 4.7W for generating the same amount of partial solutions in the NUC and Tinker Board, respectively. The path-based search approach consumes around 7.7 and 5.15 times more energy compared to the Q-learning based approach for generating the same amount of partial solutions. It is important to note that the path-based algorithm would not run on the NUC or Tinker Board due to limited resources and lack of a GPU.

The Q-learning based algorithm can solve any size CDO quickly without consuming much power. From Table 19 and 20, it is clear that this Q-learning based approach can generate the same amount of solutions at runtime as highly optimized parallel solvers while consuming less power. Our algorithm’s required time to generate all partial solutions gets higher with higher-order instances. This is however still in the acceptable range for autonomous decision making.

TABLE 19. **Generated solutions for each nine solutions set.**

Structure of each nine solution sets	Total complete solutions for each set (not generated)
abc(4096) , abc(4096) , abc(4096), nbc(4096) , nbc(4096) , nbc(4096) , abc(4096) , cbs(1) , cbs(1) , cbs(1) , abc(4096) , nbc(4096)	3.2451^{32}
abc(4096) , abc(4096) , abc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , cbs(1) , cbs(1) , cbs(1) , abc(4096) , nbc(4096)	3.2451^{32}
abc(4096) , abc(4096) , abc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , cbs(1) , cbs(1) , cbs(1) , cbs(1) , abc(4096) , nbc(4096)	7.9228^{28}
abc(4096) , abc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , abc(4096) , cbs(1) , cbs(1) , cbs(1) , abc(4096) , nbc(4096)	3.2451^{32}
abc(4096) , abc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , cbs(1) , cbs(1) , cbs(1) , abc(4096) , nbc(4096)	3.2451^{32}
abc(4096) , abc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , cbs(1) , cbs(1) , cbs(1) – cbs(1) , abc(4096) , nbc(4096)	7.9228^{28}
abc(4096) , abc(4096) , cbs(1) , nbc(4096) , nbc(4096) , nbc(4096) , abc(4096) , cbs(1) , cbs(1) , cbs(1) , abc(4096) , nbc(4096)	7.9228^{28}
abc(4096) , abc(4096) , cbs(1) , nbc(4096) , nbc(4096) , nbc(4096) , nbc(4096) , cbs(1) , cbs(1) , cbs(1) , abc(4096) , nbc(4096)	7.9228^{28}
abc(4096) , abc(4096) , cbs(1) , nbc(4096) , nbc(4096) , nbc(4096) , cbs(1) , cbs(1) , cbs(1) , cbs(1) , abc(4096) , nbc(4096)	1.9348^{25}

TABLE 20. **Required time and energy to generate for 12 instances.**

Algorithm	Path-based forward checking	Q-learning-based ALG	
Hardware	GPU enabled System	NUC	Tinker Board
Time	8000ms	7471.5ms	69305.5ms
Energy	1680J	217.5J	325.74J

Average required time and energy consumption to generate all partial solutions for Twelve instances

E. DISCUSSION

This Q-learning-based algorithm generates partial solutions for each instance separately and concatenates them to construct complete solutions. The total number of partial solutions varies from one to one-million based on the assertions, constraints, and flag settings (FF, FT, TF). The Q-learning based algorithm can generate millions of partial solutions for any order of instances in milliseconds (see Table 14). The rest of the time is spent on concatenating these partial solutions to construct complete solutions. For example, the algorithm generated 2,097,152 complete solutions in 12782.8ms (see Table 17), where the required time to produce partial solutions is 50ms (4.9ms$\times$10). If the requirement of creating complete solutions are ignored, as the partial solutions contain the information of every instance node, this approach becomes very efficient.

It is impossible for any of the existing solvers, except for this Q-learning-based algorithm and the parallel path-based forward checking algorithm, to generate solutions for higher-order instances at runtime currently. However, only our novel Q-learning-based solving approach can run on low-power consuming platforms and handle large search spaces in runtime. It was developed to allow a cognitive agent to mine knowledge in real-time without the need for a high-performance hybrid computing system. Different flag settings of the algorithm will allow agents to generate solutions based on the necessity to make optimum decisions.

IX. CONCLUSION

In this study we developed a novel knowledge storing and mining approach using the Q-learning algorithm to enable a cognitive agent to operate in real-time. This Q-learning-based approach stores large and complex CDOs in a Q-table in a very compact and efficient manner. The use of the Q-learning algorithm for knowledge representation has not been examined before.

The solution search space for moderate-sized CDOs with instance properties can be huge. In this study, we examined problems where about one nonillion (10^{30}) solutions exist. This huge solution search space hinders any existing CDO solvers from performing in runtime without high performance computing hardware. Only GPU-enabled, highly optimized parallel solvers can generate solutions very fast with the cost of a larger size, weight, and power-consuming system (at least 200W, and likely over 50lb).

Our novel search algorithm can generate solutions from the Q-table at runtime based on the assertions and constraints quickly on low power hardware. This algorithm generated 2,097,152 partial solutions based on specific assertions and constraints in 50ms (4.9ms x 10) in the NUC. The search algorithm spends much of its time concatenating partial solutions to construct complete solutions rather than producing them. Unlike other CDO solvers, this algorithm can generate different numbers of solutions based on the user's requirement using two flags. This work allows running CDO solving tasks on low SWAP platforms, such as the Asus Tinker Board (max power: 5W and weight of 1.94oz) or the Intel NUC (power consumption around 30W and a weight of 10.6oz). Our approach consumes around 7.7 and 5.15 times less energy to generate the same amount of solutions than the GPU-enabled optimized path-based forward checking CDO solver [11] running on the NUC and Tinker Board respectively.

As future work, we will introduce objective functions to our CDO solving approach. An objective function ranks the generated solutions based on a specific criterion that describes the decision's quality. The objective function will enable cognitive agents to make optimal decisions accurately.

REFERENCES

[1] R. Wray, R. Chong, J. Phillips, S. Rogers, and B. Walsh, "A survey of cognitive and agent architectures," Dept. Elect. Eng. Comput. Sci., Univ. Michigan, MI, USA, Jan. 2007. [Online]. Available: http://ai.eecs.umich.edu/cogarch0/
[2] H.-Q. Chong, A.-H. Tan, and G.-W. Ng, "Integrated cognitive architectures: A survey," *Artif. Intell. Rev.*, vol. 28, no. 2, pp. 103–130, Aug. 2007.
[3] J. E. Laird, *The Soar Cognitive Architecture*. Cambridge, MA, USA: MIT Press, 2012.
[4] J. R. Anderson, *The Architecture of Cognition*. Cambridge, MA, USA: Harvard Univ. Press, 1983.
[5] J. R. Anderson, *How Can the Human Mind Occur in the Physical Universe?* Oxford, U.K.: Oxford Univ. Press, 2009.
[6] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, "An integrated theory of the mind," *Psychol. Rev.*, vol. 111, no. 4, pp. 1036–1060, 2004.
[7] T. Atahary, T. M. Taha, and S. Douglass, "Hardware accelerated cognitively enhanced complex event processing architecture," in *Proc. 14th ACIS Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput.*, Jul. 2013, pp. 283–288.
[8] D. Luckham, "The power of events: An introduction to complex event processing in distributed enterprise systems," in *Proc. Int. Symp. Rules Rule Markup Languages Semantic Web*, 2008, p. 3.
[9] S. A. Douglass and W. M. Christopher, "Concurrent knowledge activation calculation in large declarative memories," in *Proc. 10th Int. Conf. Cogn. Modelling*. Philadelphia, PA, USA: Drexel Univ., 2010, pp. 55–60.
[10] *EsperTech*. Accessed: Feb. 3, 2021. [Online]. Available: http://www.espertech.com/esper/
[11] T. Atahary, T. M. Taha, and S. Douglass, "Parallelized path-based search for constraint satisfaction in autonomous cognitive agents," *J. Supercomput.*, vol. 77, pp. 1667–1692, May 2020.
[12] N. Rahman, "Low power based cognitive domain ontology solving approaches," Ph.D. dissertation, ECE, Univ. Datyon, Dayton, OH, USA, 2021.
[13] E. S. Low, P. Ong, and K. C. Cheah, "Solving the optimal path planning of a mobile robot using improved Q-learning," *Robot. Auto. Syst.*, vol. 115, pp. 143–161, May 2019.
[14] S. Li, X. Xu, and L. Zuo, "Dynamic path planning of a mobile robot with improved Q-learning algorithm," in *Proc. IEEE Int. Conf. Inf. Autom.*, Aug. 2015, pp. 409–414.
[15] C. Li, J. Zhang, and Y. Li, "Application of artificial neural network based on Q-learning for mobile robot path planning," in *Proc. IEEE Int. Conf. Inf. Acquisition*, 2006, pp. 978–982.
[16] M. Bennis and D. Niyato, "A Q-learning based approach to interference avoidance in self-organized femtocell networks," in *Proc. IEEE GLOBECOM Workshops*, Dec. 2010, pp. 706–710.
[17] J. Nie and S. Haykin, "A Q-learning-based dynamic channel assignment technique for mobile communication systems," *IEEE Trans. Veh. Technol.*, vol. 48, no. 5, pp. 1676–1687, 1999.
[18] J. Ni, M. Liu, L. Ren, and S. X. Yang, "A multiagent Q-learning-based optimal allocation approach for urban water resource management system," *IEEE Trans. Autom. Sci. Eng.*, vol. 11, no. 1, pp. 204–214, Jan. 2014.
[19] A. Rahimi-Kian, B. Sadeghi, and R. J. Thomas, "Q-learning based supplier-agents for electricity markets," in *Proc. IEEE Power Eng. Soc. Gen. Meeting*, 2005, pp. 420–427.
[20] R. Budiu, "ACT-R," ACT-R Research Group, Dept. Psychol., Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. 2002-2013, 2013. Accessed: Jun. 6, 2020. [Online]. Available: http://act-r.psy.cmu.edu/about/
[21] John Laird's Research Group at the University of Michigan. (2020). *Soar Cognitive Architecture*. Accessed: Jun. 6, 2020. [Online]. Available: https://soar.eecs.umich.edu/
[22] J. E. Laird, K. R. Kinkade, and S. Moha, "Cognitive robotics using the soar cognitive architecture," in *Proc. Workshops 26th AAAI Conf. Artif. Intell.*, 2012, pp. 46–54.
[23] J. M. Siskind, "Screaming Yellow Zonkers," MIT Artif. Intell. Lab., Cambridge, MA, USA, Sep. 1991.
[24] T. Atahary, T. M. Taha, and S. Douglass, "Parallelized mining of domain knowledge on GPGPU and Xeon Phi clusters," *J. Supercomput.*, vol. 72, no. 6, pp. 2132–2156, Jun. 2016.
[25] T. Atahary, T. Taha, F. Webber, and S. Douglass, "Knowledge mining for cognitive agents through path based forward checking," in *Proc. IEEE/ACIS 16th Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput. (SNPD)*, Jun. 2015, pp. 1–8.

[26] N. Jussien, G. Rochart, and X. Lorca, "Choco: An open source Java constraint programming library," in *Proc. Workshop Open-Source Softw. Integer Contraint Program. (CPAIOR)*, 2008, pp. 1–10.

[27] T. Atahary, T. Taha, and S. Douglass, "Parallelizing knowledge mining in a cognitive agent for autonomous decision making," in *Proc. Comput. Conf.*, Jul. 2017, pp. 1058–1068.

[28] M. Edmonds, T. Atahary, S. Douglass, and T. Taha, "Hardware accelerated semantic declarative memory systems through CUDA and MapReduce," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 3, pp. 601–614, Mar. 2019.

[29] M. Edmonds, T. Atahary, T. Taha, and S. A. Douglass, "High performance declarative memory systems through MapReduce," in *Proc. IEEE/ACIS 16th Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/Distrib. Comput. (SNPD)*, Jun. 2015, pp. 1–8.

[30] N. Rahman, T. Atahary, T. Taha, and S. Douglass, "Cognitive domain ontologies on the TrueNorth neurosynaptic system," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, May 2017, pp. 3543–3550.

[31] N. Rahman, T. Atahary, T. Taha, and S. Douglass, "A pattern matching approach to map cognitive domain ontologies to the IBM TrueNorth neurosynaptic system," in *Proc. Cognit. Commun. Aerosp. Appl. Workshop (CCAA)*, Jun. 2017, pp. 1–4.

[32] L. Cao, H. Zhang, Y. Zhao, D. Luo, and C. Zhang, "Combined mining: Discovering informative knowledge in complex data," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 41, no. 3, pp. 699–712, Jun. 2011.

[33] D. Fisch, E. Kalkowski, and B. Sick, "Knowledge fusion for probabilistic generative classifiers with data mining applications," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 3, pp. 652–666, Jan. 2013.

[34] S. Nirkhi, "Potential use of artificial neural network in data mining," in *Proc. 2nd Int. Conf. Comput. Autom. Eng. (ICCAE)*, vol. 2, Feb. 2010, pp. 339–343.

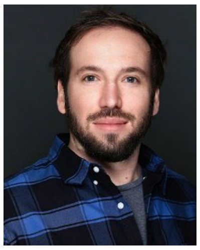
[35] L. Cao, "Domain-driven data mining: Challenges and prospects," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 6, pp. 755–769, Jun. 2010.

[36] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, M. Hasan, B. C. Van Essen, A. A. S. Awwal, and V. K. Asari, "A state-of-the-art survey on deep learning theory and architectures," *Electronics*, vol. 8, no. 3, p. 292, Mar. 2019.

[37] A. Ngo. (2018). *Intel NUC Kit NUC8i7BEH (i7-8559U) Mini PC Review*. [Online]. Available: https://www.notebookcheck.net/Intel-NUC-Kit-NUC8i7BEH-i7-8559U-Mini-PC-Review.360356.0.html

NAYIM RAHMAN (Member, IEEE) received the B.S. degree in electrical and electronics engineering from the Rajshahi University of Engineering and Technology (RUET), Bangladesh, in 2008, the M.S. degree in electrical and electronics engineering from Wright State University, Dayton, OH, USA, in 2012, and the Ph.D. degree in electrical and computer engineering from the University of Dayton, OH, in 2021. He is a Research Engineer with the University of Dayton. His research interests include low power-based optimal decision-making and spiking-based neuromorphic programming on non-von Neumann devices.

TANVIR ATAHARY received the B.S. and M.S. degrees from the University of Dhaka, Bangladesh, in 2006 and 2008, respectively, and the Ph.D. degree from the University of Dayton, in 2016. Before starting his Ph.D. study, he was a full-time Faculty Member with the University of Liberal Arts (ULAB), Bangladesh, from 2008 to 2011. He has been closely working with Wright Patterson Air Force Base (WPAFB) from his first year of Ph.D. study and spent consecutive five summers with WPAFB. After completing the Ph.D. study, he joined the University of Dayton, as a Research Engineer and WPAFB, as a full-time Contractor. His research interests include high performance computing and cognitive computing architectures. Besides performing research he teaches undergraduate students with the University of Dayton, as an Adjunct Faculty, since 2018.



CHRIS YAKOPCIC (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering and the master's degree in business administration from the University of Dayton (UD), in 2009, 2011, 2014, and 2022, respectively. He is currently on the Research Faculty of UD. He works on developing algorithms for spiking neural network processors and porting deep learning to low power embedded systems. His current research interests include memristor device modeling, analog circuit design with memristor devices, and implementing neuromorphic algorithms on memristor crossbars. In 2013, he received the IEEE/INNS International Joint Conference on Neural Networks Best Paper Award for a paper on memristor device modeling. In 2019, he was chosen as the IEEE Dayton Section Computer Society Award winner for his work on memristor based electronic systems for extreme low-power computation and cutting edge algorithms for autonomous systems. In 2022, he received the Best Paper Award at the ACM International Symposium on Nanoscale Architectures.

TAREK M. TAHA (Senior Member, IEEE) received the B.S. degree from DePauw University, Greencastle, IN, USA, in 1996, and the B.S.E.E., M.S.E.E., and Ph.D. degrees in electrical engineering from the Georgia Institute of Technology, Atlanta, in 1996, 1998, and 2002, respectively.

He is a Professor with the Electrical and Computer Engineering Department, University of Dayton. His research was supported by multiple agencies and companies, including the National Science Foundation, the Air Force Research Laboratory, and the National Aeronautics and Space Administration. His research interests include cognitive computing architectures, high performance computing, and architectural performance modeling. He is a member of the IEEE Computer Society. He received the NSF CAREER Award, in 2007.

SCOTT DOUGLASS received the Ph.D. degree from Carnegie Mellon University. He is currently a Senior Cognitive Scientist with the AFRL Aerospace Systems Directorate, Control Systems Branch. He leads basic and applied research efforts in a portfolio developing agile mission planning and contingency management capabilities for autonomous platforms. His current research interests include problem solving as SAT/SMT model checking, multi-criteria decision analysis, information/knowledge extraction from unstructured language, and approximate SAT solving using neuromorphic computing devices.