A Thesis

entitled

Performance Enhancement of Data Retrieval from Episodic Memory in Soar Architecture

by

Man B. Bhujel

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the Masters of Science Degree in Electrical Engineering

Dr. Vijay Devabhaktuni, Committee Chair

Dr. Ahmad Javaid, Committee Co-Chair

Dr. Devinder Kaur, Committee Member

Dr. Amanda Bryant-Friedrich, Dean
College of Graduate Studies

The University of Toledo
May 2018

An Abstract of

Performance Enhancement of Data Retrieval from Episodic Memory
in Soar Architecture

by

Man B. Bhujel

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the
Masters of Science Degree in Electrical Engineering

The University of Toledo
May 2018

Episodic memory has been the key component of various intelligent and cognitive architecture that includes the autobiographical events of past experiences. The implementation of episodic memory enhances the performance of cognitive agents by utilizing past history for decision making. During episodic retrieval in Soar architecture, the cue matching step involves a two stage process to improve the performance of the architecture. Soar implements cue matching as a surface cue analysis, which finds candidate episodes based on the matched leaf node. It then performs a structural match on candidate episodes with full surface match. However, continuous design research is still needed for minimizing the operational time of episodic processes along with timely cue-matching as episodic memory grows.

This thesis provides an insight to improve on both stages of cue matching, which ultimately leads to a quick retrieval of episodes. First, the approximations of original implementation base-level activation (BLA) is implemented for determining the feature weight for surface cue analysis. These methods are computationally efficient. Second, a new approach of solving the constraint satisfaction problem (CSP), arc-consistency algorithm is implemented for the structural cue analysis. For the experiment, two of the most frequently used testing environment, Eaters and TankSoar, are chosen.

From the first experiment, it is found that the Eaters agent provides comparable performance with approximations of BLA and demonstrate applications of approximations. The approximation of BLA has high computational efficiency. Determining activation value of working memory elements (WMEs) is a part of cue matching; hence, incorporating approximation of BLA leads to faster retrieval of episodic memory. Also, using a new technique for structural graph match, this thesis obtains less query time compared to original one. This also leads to decrease in retrieval time. Moreover, the results show that as the size of episodic memory increases, the rate of change of retrieval time with arc-consistency decreases. With these results, the ultimate goal of the research is achieved.

To my grandparents and parents for their endless support.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor Dr. Vijay Devabhaktuni and my co-advisor Dr. Ahmad Javaid for the continuous support of graduate study and research, for all their patience, motivation, enthusiasm, and immense knowledge provided. Their guidance helped me in all the time of research and writing of this thesis. I would also like to thank Dr. Devinder Kaur for agreeing to serve as member of my thesis committee.

Second, I would like to thank The University of Toledo's EECS Department as well as the Dean's office for providing me with financial support in the form of graduate assistantship and tuition waivers. My sincere thanks also goes to Dr. Richard Molyet, for enriching my graduate experience with teaching assistant opportunities. I would like to extend my sincere gratitude to the Air Force Research Laboratory (AFRL) for providing the EECS Department with the funding for the research portion of my assistantships. I especially want to express my appreciation to all SoarTech team members especially Jack Zaientz and Robert P. Marinier III for their suggestion throughout this research work. I also want to thank Student Disability Services Department of The University of Toledo, notably John Satkowski, for providing me with part-time employment during the first Fall semester of my graduate studies.

In addition, I thank my lab colleagues for assisting during my graduate research, stimulating discussions, and editing my thesis. My research group members of NE2042 and NE1024 are also greatly acknowledged. Lastly, I would like to thank my family for continuous love, support, and motivation throughout my graduate studies.

Contents

List of Tables

List of Figures

List of Abbreviations

AC	Arc-Consistency
ACT-R	Adaptive Control of Thought-Rational
AI	Artificial Intelligence
API	Application Program Interface
BT	Backtracking
CBR	Case Based Reasoning
CSP	Constraint Satisfaction Problem
DFS	Depth First Search
GT	Generate and Test
MAC/FAC	Many-are-Called/Few-are-Chosen
NC	Node-Consistency
NN	Nearest Neighbor
PC	Path-Consistency
Soar-EpMem	Soar Episodic Memory
SME	Structural-Mapping Engine
WM	Working Memory
WMA	Working Memory Activation
WME	Working Memory Element
WMG	Working Memory Graph

List of Symbols

A	WME Attribute
cn	constraint network
C	a set of constraints of CSP
D	a set of domains for each variables
E	any particular episode in episodic memory
I	WME identifier
ϕ	natural projection to map WMEs
Q	the cue provided for the episodic memory retrieval
Rel_{ij}	a subset of Cartesian product of two domains representing allowed pairs of values
V	WME Value
Var	ordered variables of CSP
W	a set of WMEs
X	a set of variables of CSP

Chapter 1

Introduction

Most of the recent cognitive systems have been developed in order to meet human level performance. One advantage of human over artificial intelligent (AI) systems is that humans are able to remember personal experiences and use them to learn or improve future decision making. This type of memory is first described by Endel Tulving in 1972 as "Episodic Memory" and characterized the distinction between knowing and remembering [1]. The ability to remember where, what, when, and other actions that have taken in various situations provides information for acting in the present. Knowing the history facilitates to perform several cognitive capabilities in context of sensing, reasoning and learning.

The simple way of getting the benefits from an agent's own experiences is to record them so then later when appropriate they can be examined and reexamined. Episodic memory can facilitate that process and ultimately improve the cognition. Cognitive architecture can use its episodic memory to evaluate the resource cost versus gain of taking certain actions. For example, when an agent meets an enemy, the agent can examine the past success and failure of possible actions that resulted from a similar situation and decide the best way to engage the enemy.

Previously, episodic memory was considered to be less important than semantic memory, but recent research has found that an episodic memory has been crucial in

supporting many cognitive capabilities. Rather than going through an entire procedure, the system can directly access the episodic memory to see past events that resembles the current situation and takes an action by analyzing the past outcomes. Moreover, the representation of events in the fashion of spatio-temporal, short-term slices with a record of progress in goal processing helps to analyze the performance after the operation.

Episodic memory cannot be isolated, and its performance cannot be analyzed solely. Instead, it is a component of cognitive architecture that can effectively exploit the functionality of episodic memory. For this research, Soar architecture is selected as it is used for both cognitive modeling and artificial intelligence research. Soar also shares the features of ACT-R [2, 3], Icarus [4], Clarion [5], and EPIC [6] architectures.

Although, Soar architecture has effective and efficient implementation of episodic memory, which is presented in [7, 8], there is always a need to minimize the growth in processing time for an episodic memory operations as the number of episodes increases.

1.1 Soar Cognitive Architecture

Soar cognitive architecture has been developed since 1983 [9] to build general intelligent systems. Soar includes a full range of problem solving methods, interact with the outside world, and learn about all aspects of tasks along with its performance. The major principles of Soar architecture design is to have distinct architectural mechanisms represented as productions (permanent knowledge), objects (temporary knowledge), automatic sub-goaling, and learning mechanisms. Moreover, it includes procedural, semantic, and episodic memory as well as their respective learning processes as reinforcement, semantic, and episodic learning to enhance the Soar agent performance [10]. Fig. 1-1 displays the overview of the Soar architecture.

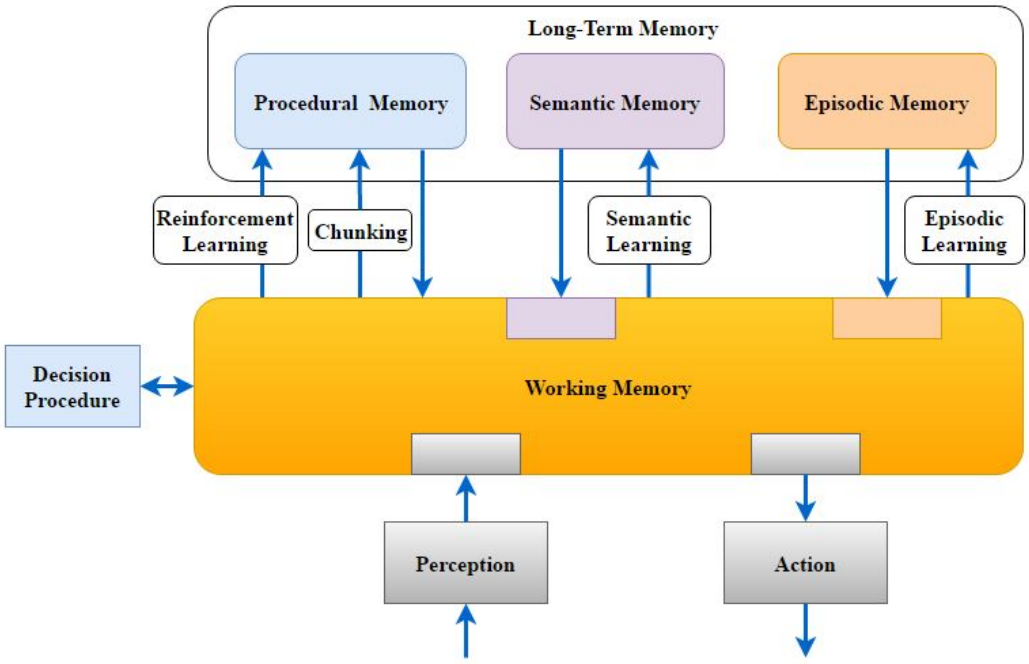


Figure 1-1: Memory structure in Soar cognitive architecture.

Soar models cognition is based on the symbolic production rule system as a sequence of decision cycles. Soar stores its current knowledge and modifies them according to goal in temporary memory called working memory (WM). WM reflects the features, defined by the symbolic triple (identifier, attribute, value), also known as working memory elements (WMEs) available via sensing and internal processing, whereas case-based consists of the pre-specified cases and their outcomes [11]. This working memory depicts the graphical structure known as working memory graph (WMG) [8] as shown in Fig. 1-2.

The same graphical structure is followed by episodic memory in Soar. This makes it possible for effective and efficient implementation in terms of episodic storage, encoding, and retrieval. As working memory holds the short-term knowledge, it is possible to capture the snapshot by episodic memory for the history of experience. This makes the Soar-EpMem task-independent, automatic, and architectural [8].

Everything present in the working memory is stored as an episode in episodic

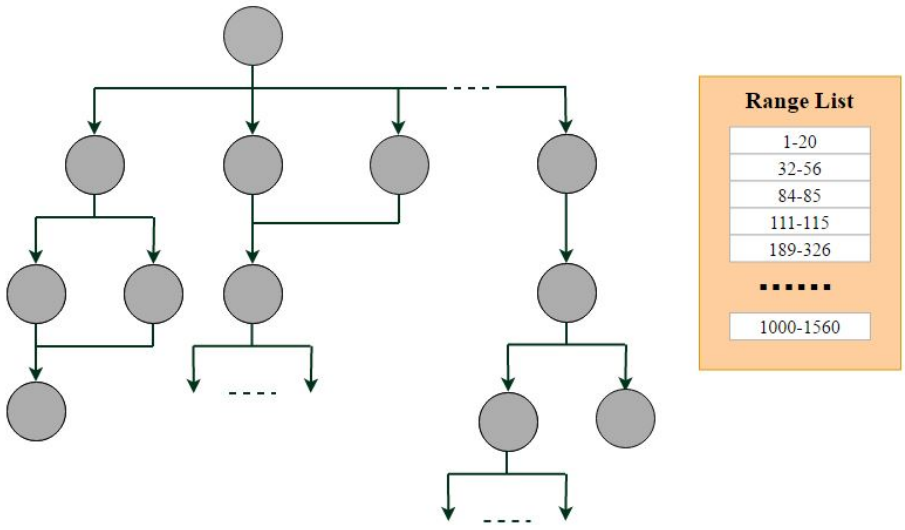


Figure 1-2: Typical working memory graph structure implemented in Soar and temporal intervals of a node captured in episode.

memory. This memory is queried by creating the cue in a cue generator area of working memory. The best matched episode to the cue is determined and redeemed in the retrieval area of working memory. This retrieved episode is utilized for future decision making. In a cognitive architecture, the accuracy of retrieving a episode from the episodic memory pool is always a challenging task.

1.2 Episodic Memory Retrieval

Retrieval of memory is re-accessing of information of the past from stored memory. Retrieval generally occurs in response to a particular event. In common practice, it is known as remembering and quite different from the act of thinking. These retrievals are not quite identical to the originals but are mixed with an awareness of current situations.

For accessing the memory, two main methods are involved. These methods are recognition and recall [12, 13]. Recognition is the process of comparison of information with memory based on the current event or an object. Although the memory retrieval

process depends on the architecture in which the process is involved, the basics remain the same for every architecture. The cue of the required information is generated and matched with the stored memory. The matched information is regenerated in the working memory, which is referred to as recall. Fig. 1-3 shows an overview of the episodic memory process implemented in Soar.

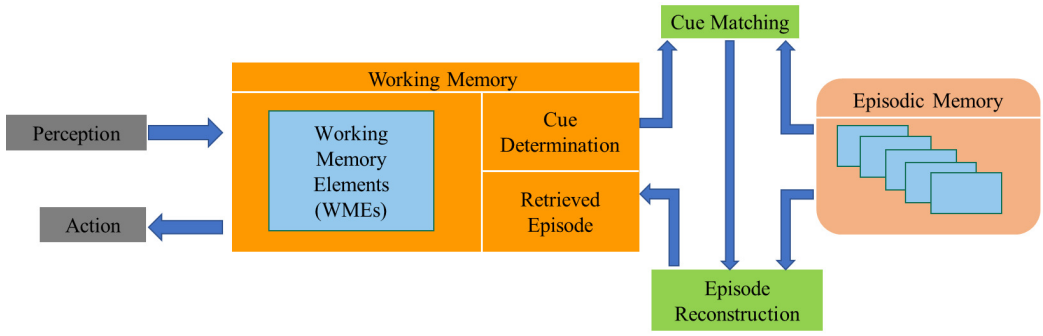


Figure 1-3: An episodic memory process in Soar.

The main process of any architecture is working memory, which holds the current state and operators reflecting the current knowledge of the world and the status in problem solving. Working memory contains elements called working memory elements (WMEs). Each WME contains a certain information. These WMEs play an important role in determining the unique episodes and also in matching the relevant episodes from the episodic memory. Here, the activation of the WMEs comes into account for finding the episode with the maximum weight. Once candidate episodes are determined, graph match is introduced to find the exact match.

In Soar, the graph matching step is defined as the constraint satisfaction problem (CSP). Soar uses a backtracking algorithm as the CSP solver. The main issue of this algorithm is the occurrence of multiple dead-ends, so it needs to perform frequent backtracking. Hence, the appropriate CSP solver algorithm is required to decrease the possibility of facing dead-ends as well as the time required for cue-matching.

1.3 Constraint Satisfaction Problem

Constraint satisfaction problems (CSPs) are mathematical problems defined as a set of given variables together with a finite set of possible values whose state must satisfy a number of limitations (constraints) [14]. CSPs are the subject of research for both artificial intelligence and operations research, as their formulation provides a common basis to analyze and solve problems. CSPs are combinatorial in nature and therefore require reasonable computational expense to find a solution. Some more complex CSPs often require a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Some common examples of CSPs problems are Sudoku, Tower of Hanoi, Crosswords, Map-coloring, Graph-Matching, etc.

For this thesis, the knowledge of CSP is required during retrieval where the provided cue takes a form of acyclic graph, partially specifying the subset of an episode [8]. To perform the matching with graphical cue to the episodes in episodic memory, the structure of cue also has to be matched. This structural match is interpreted as constraint satisfaction problem.

CSPs are typically solved using a form of search. Many powerful algorithms that became a basis of current constraint satisfaction algorithms have been designed. The most commonly used techniques are; generate and test method, simple backtracking, local search, and constraint propagation [15]. Simple backtracking is a depth-first search where partial instantiations are extended by assigning values to the variables, then ensuring that all constraints are satisfied. If a constraint is violated, the search has reached a dead-end condition, therefore, must backtrack by changing assigned variables. Local search methods are incomplete satisfiability algorithms. They may find the solution of a problem, but may fail even if the problem is satisfiable. Constraint propagation modifies the CSPs that enforce them to form a local consistency [16]. First, it turns problem into simpler form to solve and second, it proves for sat-

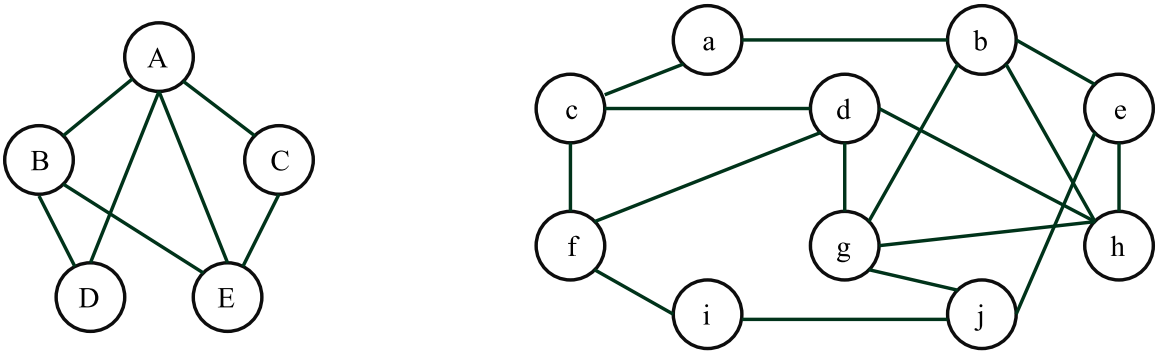
isfiability or unsatisfiability of problems. The most popular constraint propagation are arc-consistency, path consistency, and hyper-arc consistency.

Enforcing stronger consistency may lead to fewer assignments done and fewer dead-ends are explored. It is important to note that stronger methods for maintaining consistency do not produce better performance. A trade-off exists between the number of assignments performed and the computational efforts following each assignment [16].

1.4 Example of Graph Matching as CSP

One of the applications of CSP is graph matching through CSP, which is similar to this research. A simple example of graph-matching is presented. For any given two graphs: $G1$ and $G2$, the task of graph matching is to check whether $G2$ contains the subgraph that matches $G1$. There exists a graph match if:

- Every node of $G1$ can be mapped to distinct node of $G2$, and
- For all x_1, y_1 nodes in $G1$ and x_2, y_2 nodes in $G2$ are mapped respectively, whenever (x_2, y_2) is an edge in G_2, then (x_1, y_1) should be an edge in G_1



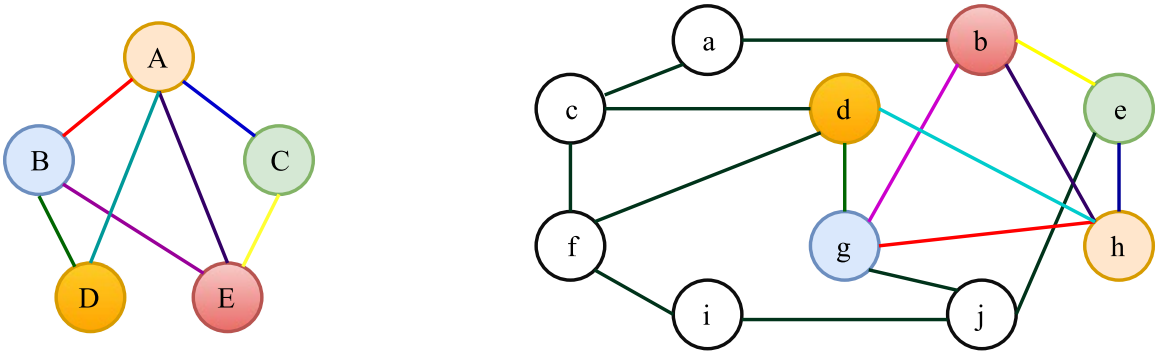
(a) Graph G1 (b) Graph G2

Figure 1-4: Example of graph matching problem.

Fig. 1-4 shows two graphs $G1$ and $G2$ as an example of a matching problem. The task is to find the subgraph $G1$ in $G2$. This can be formalized as CSP as follows:

- Variables of CSP: $\{A, B, C, D, E\}$
- Domains for each variables: $\{a, b, c, d, e, f, g, h, i, j\}$
- Constraint: For all compound labels $(< x, p >< y, q >)$, if x and y are connected in $G1$, then p and q must be connected in $G2$.

In this example, the solution exists, as $(< A, h >< B, g >)$ satisfies the constraint on A and B because (g, h) is also an edge in $G2$. The solution mapping with compound label is $(< A, h >< B, g >< c, e >< D, d >< E, b >)$. This is displayed in Fig. 1-5 below.



(a) Graph G1 (b) Graph G2

Figure 1-5: Solution to graph matching problem.

1.5 Proposed Techniques

The Soar architecture implements the two step matching algorithm to reduce the search time of retrieval. The first step implements the surface cue analysis, which counts the matching elements and/or combine with the weight associated with them.

The activation value of each element at a particular instance provides the weight to the WMEs. The second step finds the structural match among the perfect surface matched candidate to cue what is explained as graph matching. CSP backtracking is used as the graph matching algorithm.

However, occurrence of multiple frequent backtracking is the main issue for the current CSP solving technique used in Soar. The expensive graph match will execute multiple times when there is a perfect match but the structural match fails. In backtracking, due to multi-valued attributes, dead-ends might be encountered. This forces the search time to rise. Thus, the objective of this research is to reduce the overall retrieval time through the implementation of a computationally efficient activation approximation technique for assigning weights to WMEs and the application of a new graph matching algorithm.

The first approach is based on implementing approximation of activation. The original implementation of Soar uses base-level activation, which uses the exact timing information of WMEs. In contrast, the approximation tends to have less information about the WMEs, thus making the approximation more memory efficient as well as computationally efficient. However, due to the loss of information, the approximation technique slightly deviates the calculation of activation value, but the performance is comparable to that of original implementation. This makes the possibility of application of approximation as a computational model.

The second approach in this thesis tries to address issues of graph matching of hitting multiple dead-ends and backtracking frequently by implementing the consistent technique of CSP solving called the arc-consistency (AC) algorithm. Arc-consistency pre-checks all constraints from the available domains so that the possibility of hitting dead-ends reduces. This leads to performing the graph matching step only once; as a result, it reduces the time consumption during search.

1.6 Thesis Organization

This thesis proceeds as follows:

Chapter 2 provides a literature review that forms the foundation of this research. It covers a variety of theoretical backgrounds pertaining to prior established research methods in the areas of episodic memory, working memory activation, and CSP.

Chapter 3 elaborates about the working memory activation scheme, a preliminary method that focuses on approximation of the base level activation and implementing them. Also, the experimental result is presented with Eaters environment.

Chapter 4 explains on the second proposed technique on graph matching, which includes the computationally efficient algorithm for solving constraint satisfaction problems. Further explanation of the simulation results are also pointed out.

Chapter 5 draws conclusive remarks on the results. And, it also discusses possible future directions in which this research can be continued.

Finally, the thesis includes an appendix that contains source code for Eaters agent, the format of TankSoar cues to be provided to retrieve episodic memory, and tables of Eaters score and query time from different TankSoar memory.

Chapter 2

Literature Review

This chapter provides a background in previously established research works and methods that forms a foundation for the research of this thesis. Included topics are: episodic memory retrieval techniques based on respective memory model, prior work on Soar episodic memory retrieval, and finally a brief concept of constraint satisfaction problem and existing approaches to solve CSPs.

2.1 Work in Related Memory Retrieval Research Fields

The retrieval from the episodic memory depends on the way the episodes are encoded and stored. Different episodic memory modules have their own memory organization leading to their very own retrieval process.

2.1.1 MAC/FAC Similarity Based Model

MAC/FAC is a model of similarity-based retrieval that has been capable to model the interesting patterns found in psychological data [17]. The MAC/FAC implements a two-stage process for judging similarity between episodic memory pool and cues. The first stage, MAC ("many-are-called") uses a computationally cheap non-

structural matcher to constrain the size for the second stage candidate. Forbus et al. [17] implement task specific content vectors that are efficiently generated at the time of episodic storage. During query, a content vector is generated based on cue and compared with all memory item vectors. The matching candidates are subjected to the second stage, FAC ("few-are-chosen"). The FAC stage uses a Structural-Mapping Engine (SME) for structural comparison between candidate episodes and the cue. MAC/FAC model performs well in querying a small number of feature-rich and structured memory items. The experimental setup has only 32 feature-rich episodes.

Although an episode for any cognitive agent may be less complex with low feature-contents, the agent may generate thousands to millions of episodes. Also, the major goal of MAC/FAC is applicable in the psychological phenomena and may include inaccuracies in retrieval during surface and structural similarity. This model may be unsuited for the cognitive agents, particularly in cases in which the optimal result is the main concern and the size of the memory pool is large. Beside this incompatibility of this model, the method of structural matching for the retrieval from the few candidates is quite impressive and is applied in several other retrieval modules.

2.1.2 EM-A Generic Memory Module for Episodic Memory

EM [11] is a generic store to support episodic memory functionality such as planning, classification, and goal recognition. EM serves as an external component with an application program interface (API). An episode in EM is represented as a combination of three dimensions: context, contents, and outcome. Context is a general setting in which the episode happened. Contents is the ordered set of events that make up episodes. Outcome refers to the specific evaluation of the episode's effect. Episodes are stored in memory unchanged and are indexed for retrieval. Multi-layer indexing is used in the following way: i) shallow indexing in which each episode is

indexed by features and, ii) deep indexing in which episodes are linked together by how they differ structurally from one another.

During retrieval, EM cues are defined as partial episodes. This memory module employs a two-stage evaluation scheme as in MAC/FAC, whereby a constant number of potential matches are found, which are then compared using the relatively expensive semantic matching. First, shallow indexing selects a set of episodes based on the number of common features between them and the cue. Second, the hill climbing algorithm is used in this module as semantic matching, which employs sub-graph isomorphism and transformation rules in order to resolve the mismatches between two representations. This indexing mechanism and the search-based retrieval set the approach apart from serial search. The work done by Tecuci and Porter [11] was used for learning in short, single-task domains and is unclear about scalability.

2.1.3 Case Based Reasoning (CBR)

In the case-base reasoning module, the case structure is predefined and do not possess the temporal feature of episodic memory. The case-base sizes are either fixed or grow at a limited rate, which is in contrast to the episodic store that grows with experience over time. The main focus of this module is to optimize the task performance for a given case-base, where each case is a problem and its solution [18].

Efficient nearest neighbor (NN) algorithms have been studied in CBR for qualitative and quantitative retrieval [19, 20]. The underlying algorithms and data structures supporting these algorithms depend on a relatively small static number of case/cue dimensions. This module of episodic memory does not take advantage of the temporal structure inherent to episodic memories.

Heuristics methods have been applied for retrieval efficiency [21], refined indexing [22], storage reduction [23], and unwanted case deletion [24]. It seems that many researchers achieve gains through the application of two stage cue matching. Initially,

the surface similarity is considered and then structural evaluation is applied in next step [17].

The requirement of dealing with temporal cases has been acknowledged as a signicant challenge with the CBR. Other motivating research on temporal CBR systems can be found in [25], and representing and reasoning about time-dependent case attributes is observed in [26]. However, these works fail to deal with accumulating an episodic store, nor do they have efficient implementation of the temporal cases.

2.1.4 Neural Model of Episodic Memory

Neural model of the episodic memory by Wang et al. [27] learns episodic traces in response to a continuous stream of sensory input and feedback received from an environment. This model is hierarchically joining two multi-channel self-organizing network based on the fusion Adaptive Resonance Theory (fusion ART) network. This network extracts the key events and encodes in spatio-temporal relation between events dynamically creating cognitive nodes. The fusion ART network dynamics provide a set of computational processes for encoding, recognition and reproduction of patterns.

Other applications involving neural network models of episodic memory that use associative networks to store the relation between attributes of events and episodes can be found in [28, 29]. These associative models can handle partial and approximate matching of events and episodes with complex relationships; they are limited in recalling information based on the sequential cues. Shastri [30] attempted to address these challenges by encoding events as relational structures, but it shows the insufficiency in learning complex sequential information.

Wang et al. conducted several empirical experimental evaluation showing that the model provides a good performance in encoding and recalling events and episodes even with various types of cue imperfections including noisy and/or partial patterns.

These models presented the results for a few hundred episodes and still unclear to scalability to the size of the episodic memory.

2.1.5 Episodic Memory in Soar

Efficient implementation in Soar architecture is shown in [7], providing efficient storage and retrieval of task-independent episodic memory. The episodes are every snapshot of the working memory requiring large storage. Episodic memory in Soar uses a tree structure to store the episodes instead of a linear trace involving more complex representation, which is later modified by Derbinsky [8], termed as Working Memory Graph. Each node is a working memory element (WME) with some temporal information about its occurrence. Hence, episodic memory is found to be temporally indexed. This also greatly reduces the space needed for storage of episodic memory.

For retrieval, an episodic cue represented as an acyclic graph is provided. It partially specifies a subset of an episode. Cue matching is defined to identify the most recent episode that shares the larger number of leaf nodes to cue. The retrieval of an episode in Soar involves the two-phase cue matching process. The first step involves surface cue analysis followed by the structural cue analysis similar to above models. Based on the cue provided, candidate episodes are found with at least one cue leaf element. Candidate episodes with perfect surface match are submitted to the second phase for structural graph matching. Standard CSP backtracking algorithm is applied for structural graph match. The best match is the most recent structural matching. If no perfect match occurs, then it is considered as a partial match and the recent episode with the highest number of leaf element matches is considered as the best match.

2.2 Implementation of Episodic Retrieval in Soar

This section includes the development of episodic retrieval in Soar prior to the application of two stage cue-matching (partial match and structural match).

2.2.1 Pilot Implementation

This was an initial implementation of application of episodic memory in Soar architecture. The pilot implementation had a lack of task independence and limited functionality as compared to a recent one [31]. Since this application was the first step towards the implementation of episodic memory, it worked as a guideline for the future design space and requirements. After performing certain experiments, it provided base for comparison for future implementations.

Episodic memory in pilot implementation stored each element as a Soar production (rule). Soar production represents long-term knowledge as a set of conditions and a set of actions stored in production memory. The retrieval occurred when special conditions were met, which fired the production (effectively retrieving the episodic memory). This led to the development of existing storage and retrieval functionality.

The retrieval in pilot implementation began when retrieval was triggered after the agent constructed a cue in working memory deliberately. Cue was determined by creating a sub-state specifically for this purpose, and the agent itself determined the cue contents with surface features and relation to each other. After this, the selection of episode started in which episodes were stored as a collection of Soar rules. Episodic retrieval and matching were a part of a normal Soar decision cycle. The operator proposal rule would match the cue and if a match occurred, the operator application rule would recreate the memory. The best match operator was selected through operator selector in case of multiple memory matching. In the retrieval phase, the resulting structure was a copy of an original state from which an episode

was drawn and overwrote the retrieval cue structure. For meta-data retrieval, only the numeric ID of episode was provided to the agent.

2.2.2 Baseline Implementation

The most valuable insight gained from the pilot implementation is its use as a guide for defining the space of an episodic memory system [31]. Also, the importance of autoneotic episodic memory and selection of the best matched episode are noticed instead of overwriting the current state. These requirements are introduced in Soar episodic memory and known as the base-line implementation. In a broad sense, the difference between these two implementations is an introduction to independent episodic memory storage.

This independent storage allows implementatio of a partial matching algorithm that is still present in Soar. The importance of matching comes into account for a better and faster retrieval technique. Baseline implementation includes an episodic memory learning module that records a new episode based on the agents behavior. This episodic memory stores a snapshot of working memory except the contents of cue construction and episodic memory retrieval sections.

If an agent wishes to retrieve an episodic memory, it constructs a cue that consists of WMEs in which the agent is looking in a retrieved episode. After the cue is constructed, it compares the cue to stored episodic memories and determines the best match. The selected episode is reconstructed as a copy of the original working memory in reserved space for retrieval.

The retrieval in baseline implementation can be viewed as [31]:

- Retrieval initiation: Just as in the pilot implementation, retrieval is initiated when an agent constructs a cue in working memory.
- Cue determination: A cue is constructed using rules in a reserved location in

working memory. The cue includes the content of WM and can have any number of WMEs.

- Selection: During episodic retrieval, the cue is compared to all stored episodes in order to select the best matched. In the baseline implementation, the best match is determined by the number of matched WMEs between the cue and the episode. Once an episode has been retrieved, the agent can also retrieve the next episode in temporal order via a special next command.

- Retrieval: The complete episode is retrieved in a reserved area of working memory to avoid confusion with the current state of the agent.

- Retrieval meta-data: Several types of meta-data were added to baseline implementation including time, episode number, number of matching, etc.

2.2.3 Enhancement to Baseline Implementation

Other features had been added to the baseline for efficient retrieval from episodic memory discussed in following two sections below.

2.2.3.1 Partial Matching

The baseline implementation included a new partial matching algorithm used in episodic memory selection. Effective partial matching is the major issue faced by many researchers in AI. Some approaches to partial matching technique include:

- Nearest Neighbor: In this approach, the features of the query are compared to each episode. The episode with the most features matched with the query is selected. Its processing is intensive and is also highly sensitive to the amount of irrelevant features in the instance or query in [32].

- String and Hash Matching: String matching focuses on providing a near-best match rapidly [33]. If the query and instance are presented as a string of symbols, then this method can effectively be used for partial matching. The conversion to a string, however, can result in loss of information. Another option is the hash tables if an effective hash function can be found for query and instances but the risk is losing the information [34].

- Classifier Systems: Classification techniques can also be applied to partial matching. The classifier divides on instance into categories and assigns where a query belongs. Examples of classifier algorithms applied to partial match include rule induction [35], N-Gram matching [36], and Bayesian learning [37].

- Hybrid Approach: This approach improves the partial match through the combination of a fast inaccurate approach with a slow accurate approach. In particular, classifier systems (fast approach) can be used to narrow the field for a nearest neighbor search (slow approach) [38].

Regardless of extensive search procedure, a nearest neighbor search is selected for baseline because it offers the most accuracy. Retrieving the correct memory is a critical requirement for an effective episodic memory system. First, only the nearest neighbor approach is implemented as a cardinality match.

2.2.3.2 Interval Based Matching

In this approach, the episodes are stored implicitly as series of time increments (ranges of decision cycles) of each node of a working memory tree as shown in Fig. 1-2. The ranges indicate the cycles when the associated WME was in the agents working memory. This makes the required storage linear in the number of changes to working memory instead of the number of elements in working memory [7].

Then modification in the matching algorithm to use this structure is implemented as well as the usage of activation values from the episodic memory cue is used to assign weight to the features of the candidate episodes. The detailed algorithm of cue-matching using activation values is as follows [31]:

- The system traverses the episodic memory cue and the WMG and the corresponding entry in the WMG is located.
- Each entry that is matched in the WMG contains a list of ranges. Each range in these lists is assigned a match score equal to the activation level of the associated cue elements.
- All of the selected lists are merged together into a single list of ranges. If two ranges partially overlap, they are split into two or more separate ranges. For example, if one list contains two ranges (5-12, 15-25) and the other list contains one range (7-18) the merged list will contain six ranges (5-6, 7-12, 13-14, 15-18, 19-25). Each range in the merged list has a match score equal to the sum of the activation levels of all the ranges that entirely covered that range.
- The merged list is traversed to locate the range with the highest match score (the best match). In case of tie, the most recent cycle is selected.
- The episode can be recreated by traversing the WMG and creating WMEs for each node that contains the selected cycle in one of its ranges.

2.3 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined as a triplet (X, D, C) where:

- $X = \{x_1, x_2, ..., x_n\}$ is the set of variables;

- $D = \{D_{x_1}, D_{x_2}, ..., D_{x_n}\}$ is the set of variables domains, where D_{x_i} is the nonempty set of domain values for the variable x_i; and

- $C = \{C_1, C_2, ...C_m\}$ is the set of constraints that apply to the variables, restricting the allowed combination of values for variables.

The solution to a CSP is an assignment of value to each variable such that all constraints are satisfied. A CSP is satisfiable or consistent if a solution exists.

We consider that the variable domains are finite sets of values. A constraint C_i is specified by:

- A scope $scp(C_i)$, which is the set of variables to which the constraint applies, and

- A relation $rel(C_i)$, which is a subset of the Cartesian product of the domains of the variables in $scp(C_i)$. Each tuple $\tau_i \in rel(C_i)$ specifies a combination of values that is allowed (i.e. supports) or forbidden (i.e. conflicts or no-goods).

The *arity* of the constraint is the size of the scope. A constraint can be unary (*arity* 1), binary (*arity* 2), or non-binary (*arity* > 2).

A binary CSP is represented as a constraint graph in which each node is labeled by the variable, and the edge between two nodes corresponds to the binary constraint, binding the variables that label the nodes connected by the edge. Unary constraint can be represented by the cycle edge.

2.3.1 General Approaches for CSP Solving

CSP solving is basically assigning the values to all the variables without violating the given constraints. Based on the given conditions the CSP can be consistent or unsatisfiability. Several techniques are available for solving CSPs. Kumar [15] and

Tsang [14] present a good overview of these techniques. CSP solving algorithms can be categorized into two approaches as discussed below.

2.3.1.1 Systematic Search

Many algorithms for solving CSPs search systematically through the possible assignments of values to variables [39]. This method guarantees the solution if it exists or finds unsatisfiability, but it takes a very long time to do so. A CSP can be solved using the Generate-and-Test method (GT) that systematically generates each possible value assignment and then it tests all generated solutions to satisfy all the constraints. GT method creates a large number of combinations equal to the Cartesian product of all variable domains.

Faced with the combinatorial explosion of GT, the idea of combining generation and test led to a BackTracking (BT) algorithm. Backtracking is a more efficient method of systematic search that relies on Depth First Search (DFS) and based on recursive BT algorithm. BT attempts to extend the partial solution towards a complete solution, by repeatedly choosing a value for another variable. Backtrack search exhaustively and systematically explores combinations of values for variables, constructively building consistent solutions. The space requirement of BT is linear in the number of variables. There are many factors that can affect the performance of search in BT. Some are listed below:

- Thrashing: repeated failure due to same reason;
- Redundancy: conflicting values of variables are not remembered;
- Late detection of conflict: conflict is not detected before it really occurs

To overcome these shortcomings, many modifications to BT such as value ordering, heuristics for variable, forward-checking and backward-checking have been

introduced to improve its performance. The common wisdom is to instantiate the most constrained variable first [40–42].

2.3.1.2 Consistency Techniques

To improve the cost of search, it is always beneficial to enforce a consistency property at the pre-processing stage to prune the search space and maintaining it after the instantiation of each variable. The consistency-enforcing algorithm makes a partial solution of a small sub-network extensible to some surrounding network. The consistency technique is also known as a filtering technique [43]. Thus, consistency techniques rule out many inconsistent assignments at a very early stage, minimizing the search for consistent assignment. The consistency techniques are rarely used alone to solve CSP completely. Different consistency techniques include simple node-consistent, a very popular arc-consistency, and expensive path-consistency.

CSP is called Node-Consistency (NC) if and only if for every value V in the current domain D_X of X, each unary constraint on X is satisfied. Thus, the node inconsistency can be eliminated by simply removing those values from the domain D_X of each variable X that do not satisfy a unary constraint on X.

In case of binary CSP, there remains to ensure the consistency of binary constraints that cannot be achieved from NC. In a constraint graph, arc defines the binary constraint, hence called Arc-Consistency (AC). The arc is known to be consistent if and only if, for every value of V_i in the current domain of X_i, which satisfies the constraints on X_i with respect of value V_j in domain of X_j, as permitted by the binary constraints between X_i and X_j. The concept of arc-consistency is directional i.e. if an arc (X_i,X_j) is consistent that does not mean (X_j,X_i) is consistent [44].

A Path-Consistency (PC) implies AC where an arc is equivalent to the path between the defined nodes or variables. PC extends the consistency test to two or more arcs meaning that every arc present between the path needs to satisfied. Therefore,

the PC is stronger than arc-consistency but requires a lot of computational cost [45].

2.3.2 Applications of Constraint Satisfaction

Examples and applications of CSPs can be found in a variety of real-life problems because of its potential to model and solve these problems naturally and efficiently. Several problems of AI and industrial applications can be discussed and defined in terms of the constraint satisfaction problem model. Some of the major research has been done in the field of civil engineering [46] for traffic signal monitoring, mechanical engineering [47] in designing systems, interactive cognition [48], air traffic control, web applications [49], network security [50], protection and guarantee of personal data/privacy [51], etc. This framework has already been established in several applications like resource management and scheduling [52–54] (when time and shared resources are involved), industrial production planning [55], and vehicular applications such as vehicle routing [56] and car sequencing [57] used for the purposes of adding several features in car after building basic models. It has been also explored in the areas of graphical picture processing [58], natural language processing [59] (by restricting the roles that each word plays), temporal reasoning [60] (as events are temporally related to each other), etc. Even the CSP has been applied to the general applications like awareness [61]. Moreover, CSP can be applied for query processing in databases [62], which is similar to this thesis research of querying episodic memory.

Chapter 3

Approximation of Working Memory Activation for Surface Match Analysis

The first part of this thesis research involves the implementation of working memory activation for surface cue analysis to retrieve an episodic memory. Here, the research focuses on how the approximation of WMA affects the accuracy of retrieval. An episode retrieved through different WMA to control the agent movement direction in every action is applied, and the performance of the agent is analyzed. The plan is to reduce the time needed by working memory activation, which ultimately leads to faster data retrieval. The underlying implementation targets cases of episodic memory in the Soar architecture. This chapter proceeds with an introduction to WMA, mathematical models of activation functions, and an algorithm for retrieval. Then, there is an introduction to techniques followed by the agent environment used for evaluation. Finally, this chapter discusses the results obtained and provides conclusions on the perspective results.

3.1 Introduction to Working Memory Activation

In human memory, specific items are sensitive to frequency and recency of occurrence [63]. The computational model to keep track of these items in any cognitive system is determined by the working memory activation. These items are the elements presented in working memory called WME. This provides the activation values that denotes the importance of WMEs for any particular instance. Basically, the process of activation is assigning the weight to each WME.

Initially, the working memory activation is introduced by ACT family [2], which describes the activation of the contents of working memory. Later Ron Chong [64] implemented WMA in Soar to support forgetting in WM similar to the decay pattern in ACT-R.

The WMEs activation value in working memory of Soar architecture is updated as follows:

- WMEs receives initial fixed activation at the time of creation.
- As WMEs are tested by production, they get an activation boost.
- If any action tries to add existing WMEs, they get an activation boost.
- An activation level decay over time as defined by the computational model of activation.

During retrieval in Soar architecture, activation values obtained from the above procedure are used for biasing the cue to select the best matched episodes from memory that works as the feature weighting of each element. The algorithm involved for retrieval is provided in detail in Subsubsection 2.2.3.2.

3.2 Working Memory Activation Techniques

The original implementation of working memory activation is associated with the Base-level activation (BLA). Approximation of working memory activation can be used to reduce the computational time in cost of activation value accuracy as well as to reduce the size of the working memory during runtime [63]. The computational model involved in Soar architecture and its approximation are discussed below.

3.2.1 Base-level Activation

Base-level activation utilizes all the details of working memory elements. BLA uses the complete record with exact time stamps of every use requiring more execution time. The activation value of a certain WME is determined by

$$A = \ln \left[\sum_{j=1}^{n} t_j^{-d} \right] \tag{3.1}$$

where n is the number of occurrence, t_j is the time since j_{th} activation, and d represents the decay parameter.

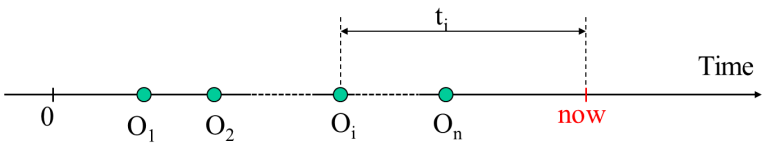


Figure 3-1: The representation of time sequence of occurrence of event for BLA.

This activation signifies the strength of memory decrease with time. More delays produce smaller losses, which is the base idea that states individual events are forgotten according to a power function.

3.2.2 Petrov's Approximation

This approximation ignores the timing of occurrence and uses two pieces of information: i) number of uses n, and ii) total lifetime t_n as shown in Fig. 3-2. The importance of new events and recency factor are diminished. It assumes all events are approximately evenly spaced. This approximation is also known as optimized learning equation.

$$A = \ln\left[\frac{n}{1-d}t_n^{-d}\right] \tag{3.2}$$

This approximation accounts for large error in determining activation value. It misses rapid, recency driven component and other transient behavioral effects.

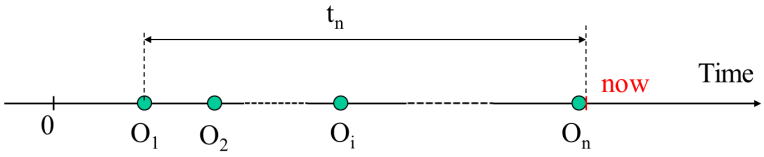


Figure 3-2: The representation of time sequence of occurrence of event for Petrov's approximation.

3.2.3 Improved Approximation

The main idea of this approximation is to keep the details of few of the most recent events and ignore the details of the distant. The new parameter, k, determines the number of recent events to be taken into account.

$$A = \ln\left[\sum_{j=1}^{k} t_j^{-d} + \frac{(n-k)(t_n^{1-d} - t_k^{1-d})}{(1-d)(t_n - t_k)}\right] \tag{3.3}$$

From the equation above, it can be seen that this function keeps track up to time stamp t_k i.e. from $j = 1$ to k where t_1 is the most recent one. Equation 3.2 is a special case of $k = 0$.

3.3 Evaluation Methodology

This section talks about the testing environment and metrics used for a performance evaluation. Initially, using MATLAB, the data having 11 features in each decision cycle for 20 time steps is generated as shown in Fig. 3-3. Then, comparison between WMAs is performed using two parameters: i) the sum of activation values obtained, and ii) the time taken for execution.

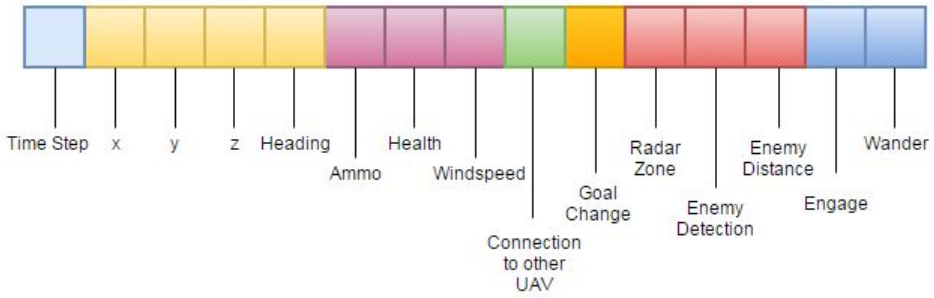


Figure 3-3: Data structure for initial WMA analysis.

After this, an agent is used to evaluate performance through the actual implementation of WMAs in architecture.

3.3.1 The Eaters Environment

To test the performance of the activation methods in real time, an episodic memory agent is created in a predefined pacman-like game named Eaters [65]. This Eaters agent moves in a grid world of 15*15 consisting of wall, normal food (blue-colored diamond shape), bonus food (red-colored square shape), or empty cell.

Fig. 3-4 shows the screen-shot of the Eaters playing board. The agent can move towards a cell in four directions except for the cell that contains a wall. If the Eaters agent moves in a cell containing food, it eats food and gains respective points, leaving the spot empty. The goal of the agent is to get the maximum points as fast as it can.

The agent senses contents of a 2-cell radius as shown in bottom right corner of Fig. 3-4.

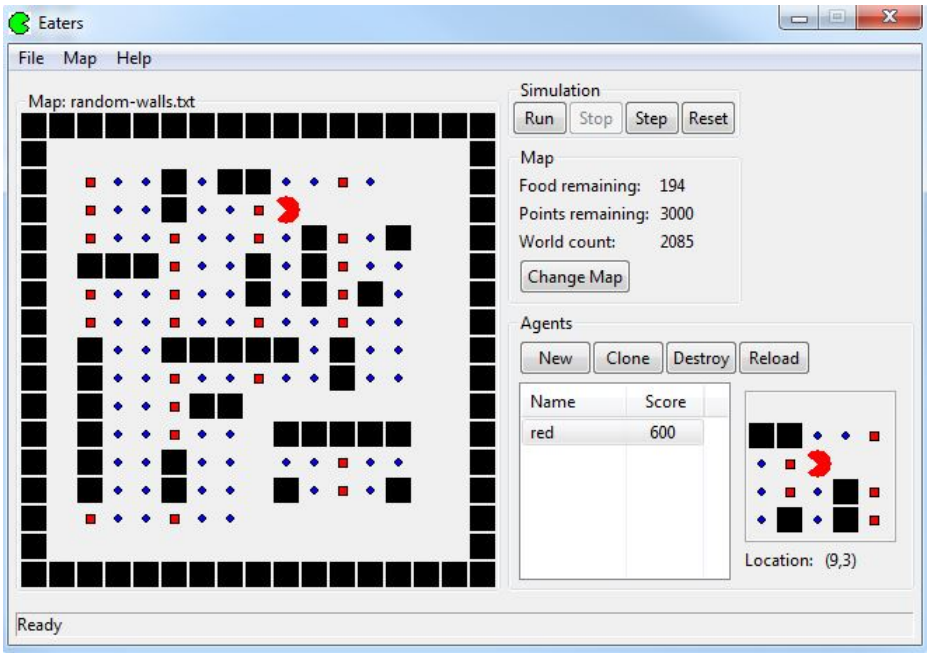


Figure 3-4: The Eaters environment.

Soar version 9.4.0 is used to test episodic memory retrieval performance. The Eaters agent has no knowledge about the value of objects it senses and does not know which direction it needs to move. The agent uses episodic memory to help it to choose which direction it should move. The agent will generate cues for each direction including internal sensing and evaluate the retrieved episodes based on the score gained moving towards a particular direction. The agent chooses the direction with the highest score gained and randomly selects if there is a tie. There is no guarantee of selecting the appropriate episode, depending upon the recorded episodes and algorithm used for retrieval. During the course of action, the agent builds up the episodic memory and explores memory for new actions.

The advantage of Eaters is that it includes a simple domain with a set of four possible objects and four possible actions for the agent, allowing users to focus on

episodic memory rather than agent and environment. This also makes users easy to gather data about performance.

3.3.2 Evaluating Methods

As an episodic memory is no use by itself, to demonstrate the performance, agents are needed in specific environments for certain tasks. Provided that the episodic memory in Soar is task-independent, action modeling in a relatively simple environment and a complex environment are used to address more difficult cognitive capabilities.

In order to evaluate the performance of episodic memory retrieval for different WMAs, an agent runs using BLA and its approximations. Then, the score gained by the agent is collected after every move for each computational model until the completion of 100 actions. Utilizing stored episodes, two more iteration are performed to study the effect of episodic memory and computational models. Based on score obtained after 100 moves and the fraction of correct evaluation done by the agent, the comparison is done among three WMAs. This measures the accuracy of retrieval and provides the feasibility of using approximations as a computational model.

3.4 Experimental Results

The preliminary result of comparison of three activation functions is shown in Fig. 3-5 and Fig. 3-6. The results include the sum of activation values of all features (Fig. 3-3) obtained from these functions and time taken for execution by each of these models.

Fig. 3-5 shows that the Improved approximation tends to follow the original Base-level activation. Although, Petrov's approximation has more error compared to that of the Improved, it follows the trend of change of activation value. Still, the performance of retrieval using these techniques needs to be analyzed. Fig. 3-6 depicts

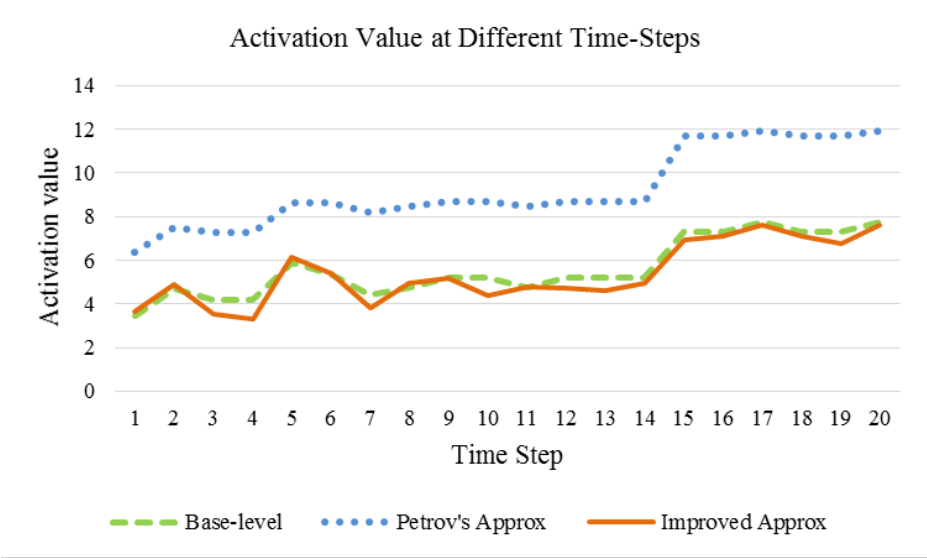


Figure 3-5: Activation dynamics under computation models.

that the Petrov's approximations of BLA is computationally the most efficient.

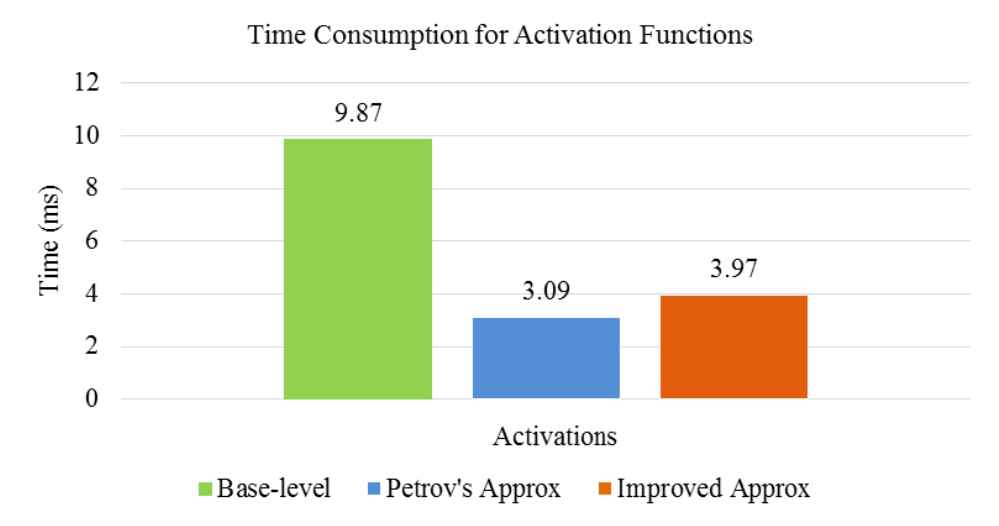


Figure 3-6: Demonstration of efficiency improvement in computation time.

For the evaluation of performance of the episodic memory retrieval, the Eaters agent is created with three different activation functions to access episodic memory pool for deciding new direction of movement. Here, biased activation to match the cue with episodes in memory is used. For each matching episode, activation values

of all matching elements are summed. The one with the highest summed activation value is considered to be the best match.

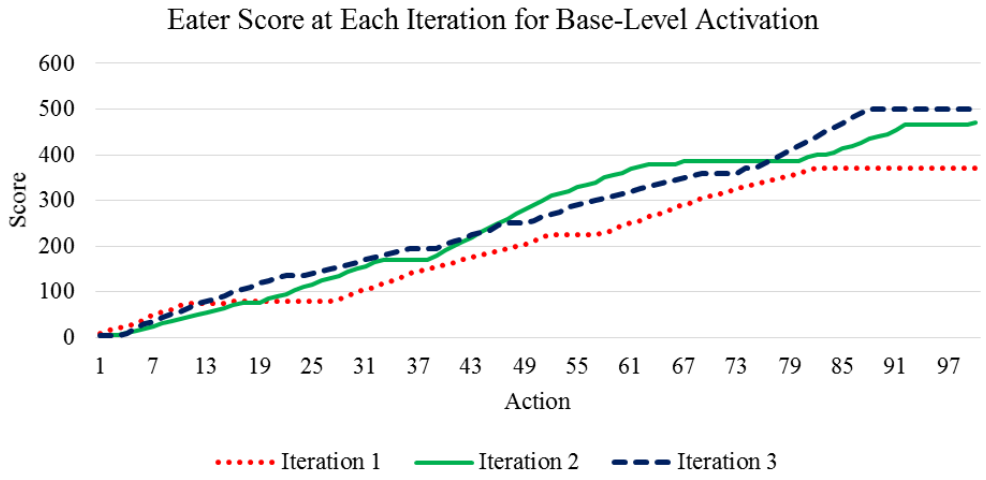


Figure 3-7: Eaters agent score depicting progress with Base-level activation.

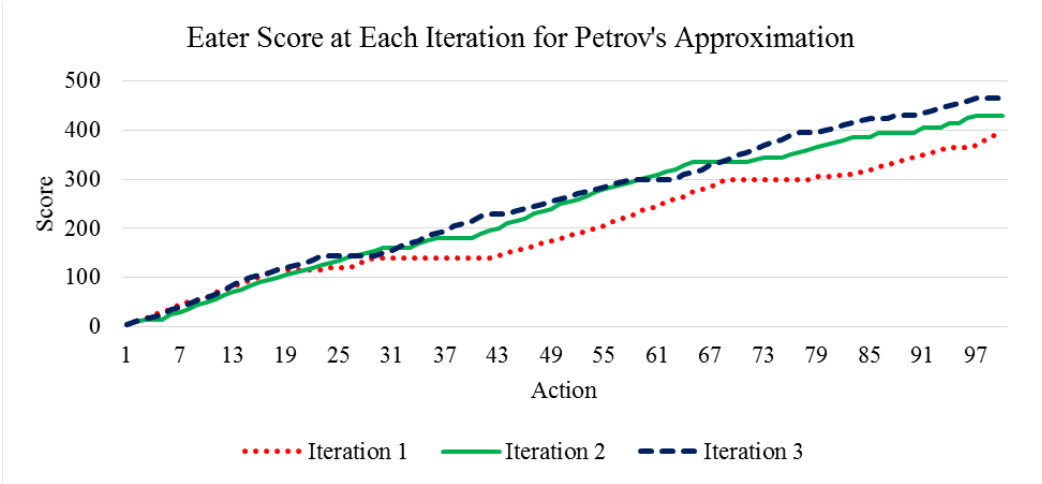


Figure 3-8: Eaters agent score depicting progress with Petrov's approximation.

Fig. 3-7, Fig. 3-8, and Fig. 3-9 depict the score of agents at each step as they progress until they reach 100 actions for three iteration using BLA, Petrov's approximation, and Improved approximation, respectively. These figures show progress of Eaters for a particular run. Moreover, the agents are learning in each step and in each

iteration using episodic memory. The results show overlapping in different iterations because of agents surrounding contents agent and the episodes that led to particular actions as mentioned earlier.

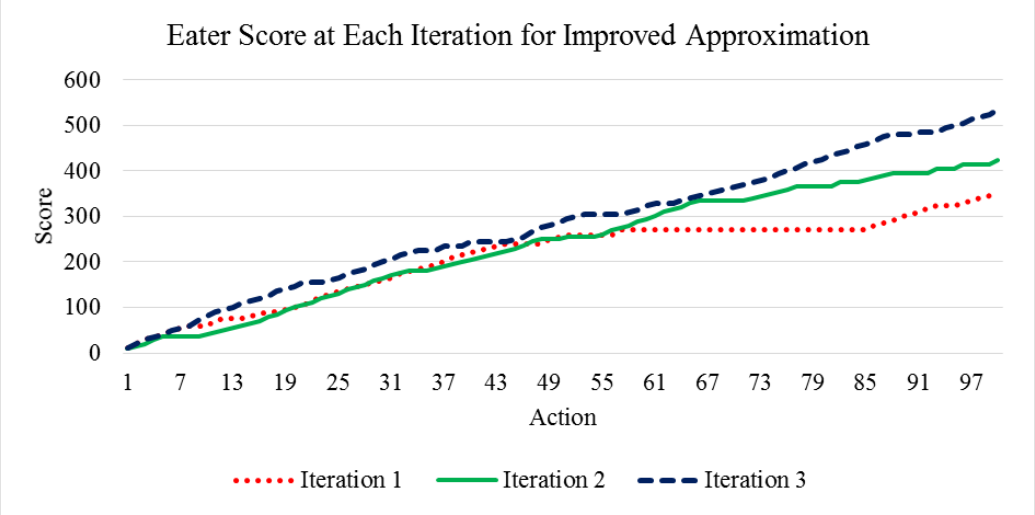


Figure 3-9: Eaters agent score depicting progress with Improved approximation.

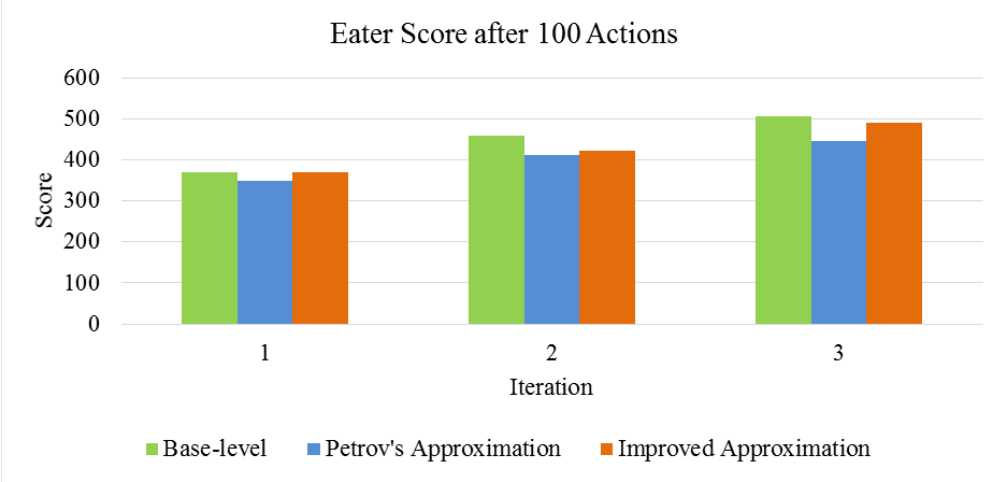


Figure 3-10: Comparison of Eaters agents score for three different activation functions.

Fig. 3-10 depicts the average final score gained by an agent after 100 moves in each iteration for each computational model. The experiment is repeated five times

for each case and scores are averaged. After the completion of 100 actions, the Eaters map and score are reset, but agent is allowed to use stored episodic memory. The plot shows that the average score of agent with BLA has the highest score in each iteration followed by Improved approximation and Petrov's approximation.

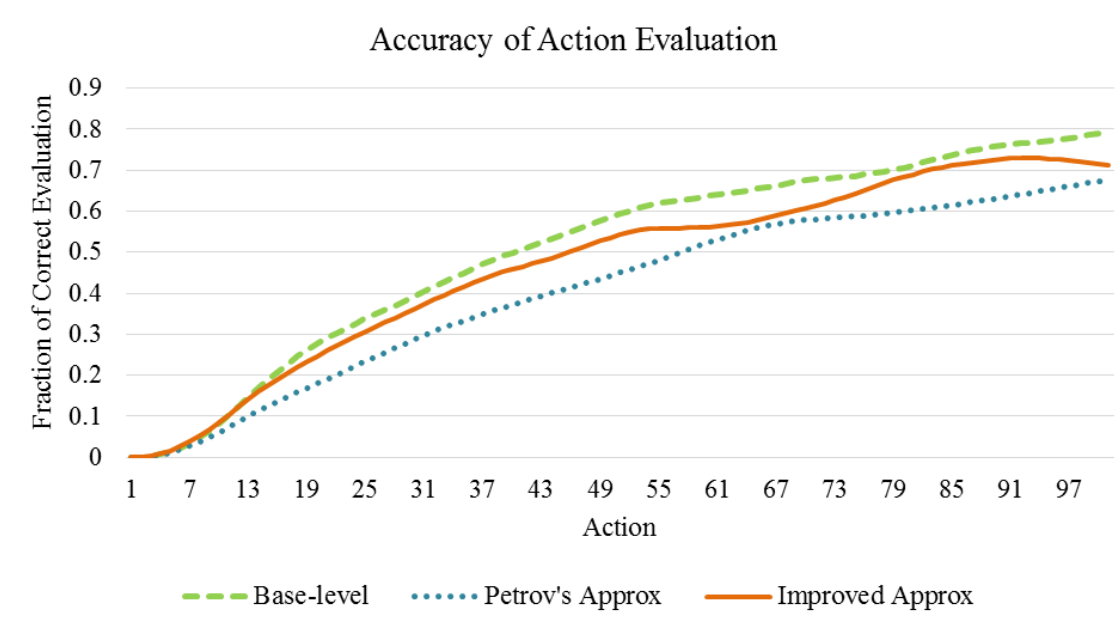


Figure 3-11: Comparison of Eaters agents learning rate for three different activation functions.

Fig. 3-11 shows an alternative view of result presented in Fig. 3-10. Fig. 3-11 shows the fraction of correct evaluation that an agent has made during the course of 100 actions. From above figure, it can be observed that the agent with Base-level activation learns the situation quite faster than agents with other two biasing techniques. Still, the result shows that there is not much difference in the learning rate of three WMAs and are comparable.

3.5 Conclusion

This chapter presents the implementation of a new model for working memory activation through the incorporation of Petrov's approximation of original activation present in Soar. The initial result shows the possibility of approximations of BLA as a computational model of working memory activation. After implementation of approximation, the Eaters agent shows the impressive progress in each iteration. Petrov's approximation as the computation model also provides the moderately good learning strategy of episodic memory. The Petrov's approximation can be applied as a substitution of original activation because the learning rates are quite comparable, as displayed in Fig. 3-11.

Petrov's approximation requires less amount of information of WMEs. That makes it computationally efficient among three activation functions by reducing the time to calculate the activation value of WMEs. As the calculation of activation value is also one of the steps required for cue-matching, incorporating this approximation method, the retrieval of episodes from memory can be reduced significantly with a slight trade-off in accuracy.

Chapter 4

Application of Arc-Consistency for Structural Graph Match during Retrieval

This chapter includes general information about structural graph matching, definition of graph matching as CSP, and two CSP solving algorithms. It also describes the environment used, methodology, results, and conclusions.

4.1 Introduction to Structural Graph Match as CSP

Structural graph match refers to finding similarity between graphs. In other words, it is finding an isomorphic image of a given graph in another graph. Recall from literature in Subsection 2.1.5, the cue provided in the Soar architecture is an acyclic graph that is to be matched with episodes in episodic memory. The second stage of cue matching involves the structural graph matching. It performs full graph matching in comparison between cue and candidate episodes structurally. This graph matching step is defined as the CSP in Soar cognitive architecture. The original algorithm

implemented in Soar involves a standard backtracking algorithm. Detailed mathematical representation of structural match as CSP is presented later in Section 4.3.

4.2 CSP Solver for Cue Matching

As mentioned the literature review in Subsection 2.3.1, various techniques are being developed to solve CSP. Among them, two CSP solving algorithms: i) backtracking and ii) arc-consistency, are popular and commonly used. Backtracking has been implemented in Soar originally. To improve performance of cue matching, this thesis implements the arc-consistency to minimize search space. The sections below describe more about these techniques in detail with the general algorithms.

4.2.1 Backtracking

Practically all complete search algorithms for CSPs are based on the backtracking algorithm. This is the basic algorithm for searching the solution of CSP. The basic operation of this algorithm is to pick one variable at a time and making sure that the variable is valid for all labels selected so far. If this is true, then the next step will be another assignment step, or termination when all variables are assigned (a solution was found). If an assignment violates a certain constraint, a backtrack step undoes the last assignment made and the next value in this unassigned variables domain is assigned instead. If the domain of the current variable is exhausted, and the next value to be assigned does not exist, another backtrack step is taken. This carries on until either a solution is found or all the combinations have been tried and failed. Since BT assigns values step-by-step and backtrack only at the last decision when it is unable to proceed, it is called chronological backtracking [14].

The algorithm of backtracking is shown below:

Algorithm 1: General backtracking algorithm

```
1  Function BT(X, D, C):Solution
2  Begin
3      BT-recursion (X, {}, D, C)
4  End
5  Function BT-recursion(unassigned_X, assigned, D, C)
6  Begin
7      If(unassigned_X is empty) Then
8          Return(assigned)
9      Else
10         Pick one value x from unassigned_X
11         Repeat
12             Pick one value v from D_x
13             Delete v from D_x
14             If assigned + new assignment < x, v > violates no constraints Then
15                 Begin
16                     Result ← (unassigned_X − {x}, assigned + < x, v >, D, C)
17                     If Result ≠ NIL Then
18                         Return Result
19                     End If
20                 End
21             End If
22         Until D_x is empty
23         Return NIL (NO Solution)
24     End Else
25 End
```

4.2.2 Arc-Consistency

Arc-Consistency measures the consistency of an arc expressed on each couple of variables of a problem with binary constraints [39]. Arc-consistency relies on a simple function called "Revise". This function is applied to a couple of variables (x_i, x_j) connected through a constraint C_{ij} by removing the locally inconsistent values from the domain D_{x_i} [43]. These couple of variables represent an arc. The Revise function deletes domain values of x_i with x_j if not compatible. In this case, the domains of x_j remains unchanged. Boolean value denotes whether the domain value is deleted or not. The arc-consistency is verified if and only if all the arcs on the constraint graph

are consistent.

Algorithm 2: General Revise Function

```
1  Function Revise(x_i, x_j):Boolean
2  Begin
3      CHANGE:=False
4      For each domain values a ∈ D_{x_i}
5          If there is no b ∈ D_{x_j} such that Rel_{ij}(x_i, x_j) Then
6          Begin
7              Delete a from D_{x_i}
8              CHANGE:=True
9          End If
10     End For
11     Return CHANGE
12 End
```

The general algorithm of AC visits every couple (x_i, x_j) and removes values from domain D_{x_i} that violate C_{ij} and if any value is removed, all the constraints are examined again, which is given by the algorithm below:

Algorithm 3: General Arc-Consistency Algorithm

```
1  Function AC(X, D, C):CSP
2  Begin
3      list:={(x_i, x_j)|C_{ij} ∈ C, i ≠ j}
4      Repeat
5          CHANGE:=False
6          For each (x_i, x_j) ∈ list
7              CHANGE:=Revise(x_i, x_j) or CHANGE
8          End For
9      Until NOT CHANGE
10     Return Consistent Network
11 End
```

4.3 Problem Formulation and Algorithms

The Soar episode is a snapshot of a working memory. Therefore, any episode has the same structure as working memory, as described in Section 1.1. Each parent

node represents an identifier of WME, a child node represents either a value or an identifier of the next WME, and the edge between nodes denotes the attribute of WME as shown in Fig. 4-1. Hence, an episode can also be defined as a directed multigraph.

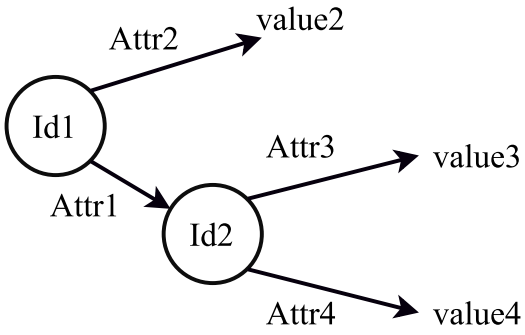


Figure 4-1: Typical WME pattern.

Let us consider a set of WME of an episode E be represented as $W = (I, A, V)$, where I, A, and V denotes the identifiers (unique symbols), attributes, and values (identifiers or any other constant values) mapped by three natural projections $\phi_I(W)$, $\phi_A(W)$, and $\phi_V(W)$, respectively.

For retrieval, a cue is provided that is partially the subset of an episode. Any cue Q contains the finite set of WMEs (W_1, W_2,... ,W_n) such that for every consecutive WME, W_i, and W_{i+1}, $\phi_I(W_i) = \phi_V(W_{i+1})$ and no two WMEs are matching.

During cue matching, an episode is considered to be fully matched to cue Q if there is a satisfaction of following conditions:

- For any $W \in E$, $E(\phi_I(W)) = Q(\phi_I(W))$
- For any $\phi_V(W) = \textit{constant}$ in Q , $E(\phi_V(W)) = Q(\phi_V(W))$
- For any $W \in E$, $E(\phi_A(W)) = Q(\phi_A(W))$
- If $\phi_I(W_{i+1}) = \phi_V(W_i)$ in E, then $\phi_V(W_{i+1}) = \phi_I(W_i)$ must be in Q

A cue Q provided for cue matching is used to generate the constraint network graph where variables are nodes, and constraints are defined by edges between nodes. These can be noted as:

- A set of variables X is a set of identifiers in cue Q
- The domain D_x of each variable x is all of the identifiers of working memory graph
- The WME with constant value in cue defines unary constraint
- Other WME in cue defines the binary constraint

The constraint network graph is generated with the algorithm below:

Algorithm 4: Constraint network generator algorithm

Input : Cue Q and working memory graph wmg
Output: Constraint Network $cn = (X, D, C)$

```
1  function ConstraintNetworkGeneration(Q, wmg);
2  for all W ∈ Q do
3      a ← Q(φ_I(W));
4      b ← Q(φ_V(W));
5      X ← X ∪ {a};
6      if b ∈ I in wmg then
           // binary constraint
7          C_ab ← edges (attribute of that particular W);
8          X ← X ∪ {b};
9          C ← C ∪ {C_ab};
10     else
           // unary constraint
11         C_a ← edges (attribute of that particular W);
12         C ← C ∪ {C_a};
13     end
14 end
15 for all a ∈ X do
       // domains
16     D_a ← I ∩ C_a;
17     D ← D ∪ {D_a};
18 end
```

The arc-consistency for structural graph match is performed with the following algorithms:

Algorithm 5: Arc-Consistency algorithm

Input : Variables order, $var = \{x_1, x_2, ..., x_n\}$ and Constraint Network, $cn = (X, D, C)$
Output: Consistent Network

```
1 function Arc-Consistency(var, cn);
2 for i ← n to 1 do
3     for j < i s.t. Rel_{j_i} ∈ cn do
          // Rel_{j_i} represent list of allowed value of pairs of values,
          /* subset of Cartesian product:  D_{x_i}, ....., D_{x_j}        */
4         Revise (x_j, x_i);
5     end
6 end
```

Algorithm 6: Revise function

Input : Variables x_i and x_j, their domains D_{x_i} and D_{x_j} and, constraint Rel_{ij}
Output: Domain D_{x_i} s.t. x_i is arc-consistent with x_j

```
1 function Revise(x_i, x_j);
2 D_{x_i} ← D_{x_i} ∩ φ_i(Rel_{ij} ⋈ D_{x_j})
```

4.4 Evaluation Methodology

This section presents the environment used for generating the episodic memory and the methods used for evaluation of structural matching. The above two approaches of CSP solver are used for full structural graph match. The total time taken for querying is measured.

4.4.1 The TankSoar Environment

TankSoar is another pre-existing two-dimensional video game environment genre known as a first person shooter. This is used to test episodic memory retrieval. In

this environment, an agent controls a tank in a 15*15 grid maze. This environment has more features than the Eaters environment. The agent plays with numerous sensors like path blockage, radar feedback, sound direction, smell, hearing, incoming path, etc. It then performs multiple actions like turn, move, attack, chase, retreat, controlling radar, shields, etc. Fig. 4-2 shows the typical TankSoar map with a display board.

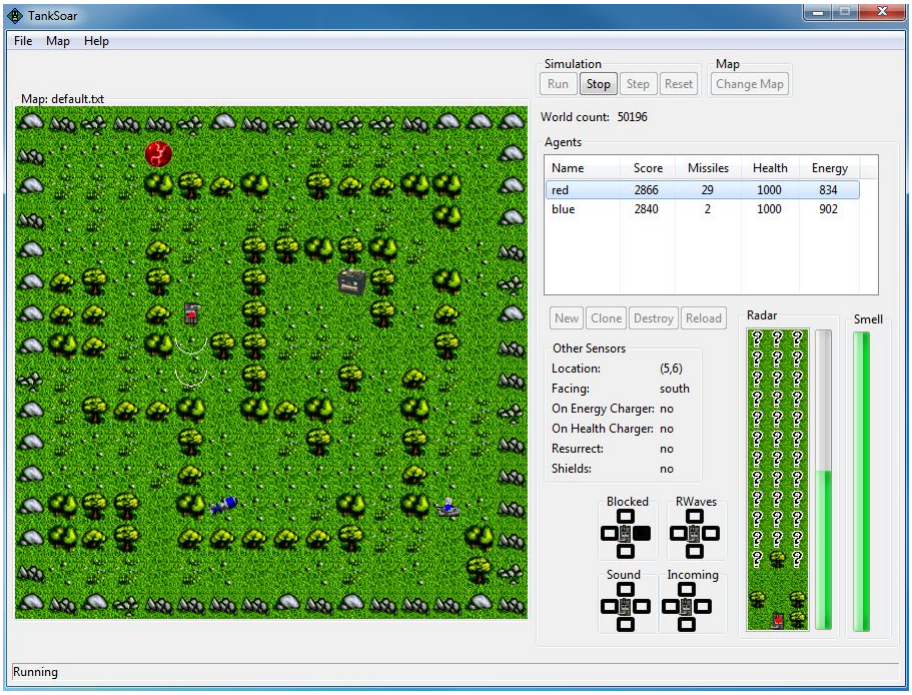


Figure 4-2: The TankSoar environment.

The TankSoar agent named *obscure-bot* is used where tank roams around the maps and tries to knock down other tanks if present. To generate the episodic memory pool, one of them populates episodes based on its sensing and action performed. This agent is run multiple times to create several sized episodic memory (size of episodic memory pools are provided in Appendix D). These memory pools are then used for the evaluation of new structural graph-match algorithm in terms of query time. For querying the generated memory, four different cues are created as described in Subsection 4.4.2.

4.4.2 Evaluating Methods

For evaluation, four different types of cues which pose questions are created as described below.

- When was I last at certain (x,y) position on my map?
- When did I last make a radar switch on?
- When did I last have full energy and health and rotate left?
- When did I last sense no blockage on left and I rotate left with radar-power setting 13?

These cues can be viewed pictorially in graphical structure as following:

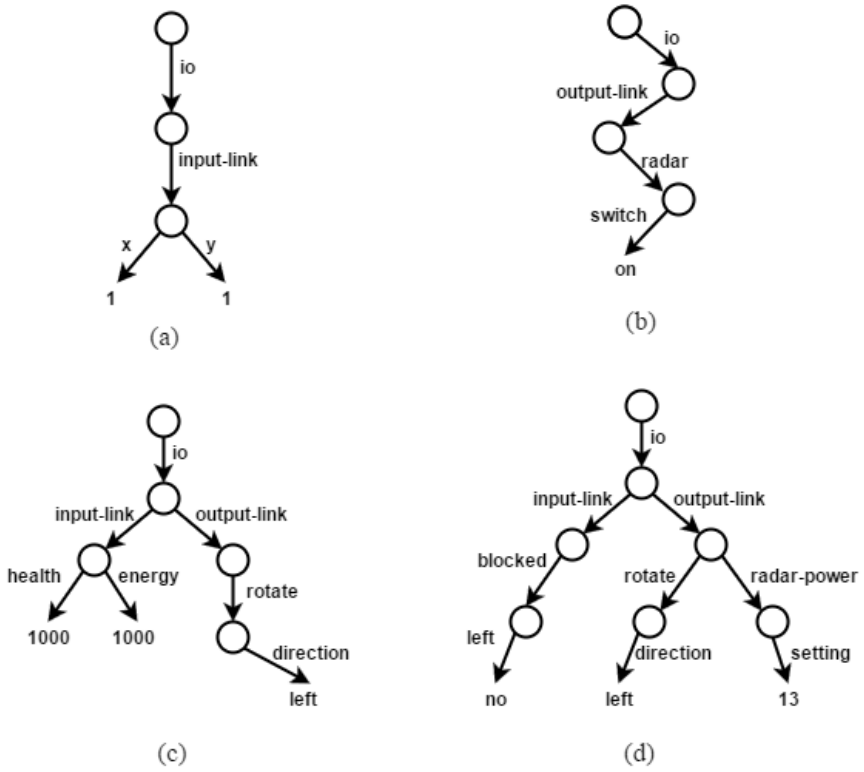


Figure 4-3: Different cue structures (a)cue 1 (b)cue 2 (c)cue 3 (d)cue 4 used to search TankSoar episodic memory.

These cues are applied to query various sized episodic memory pool generated from TankSoar using Soar architecture and the proposed algorithm. Query-time with Soar and with the proposed one is accessed. The simulation is repeated five times for each cue with each episodic memory pool and the average time taken for querying is reported for each case. The comparison is done in terms of the query-time.

4.5 Experimental Results

To evaluate new CSP algorithms, four different test cues are provided. Then, time consumed to query episodic memory by both the original Soar implementation using SoarJavaDebugger and the proposed algorithm are measured. Fig. 4-4, Fig. 4-5, Fig. 4-6, and Fig. 4-7 show the query time results which compares the current implementation of Soar to the proposed algorithm.

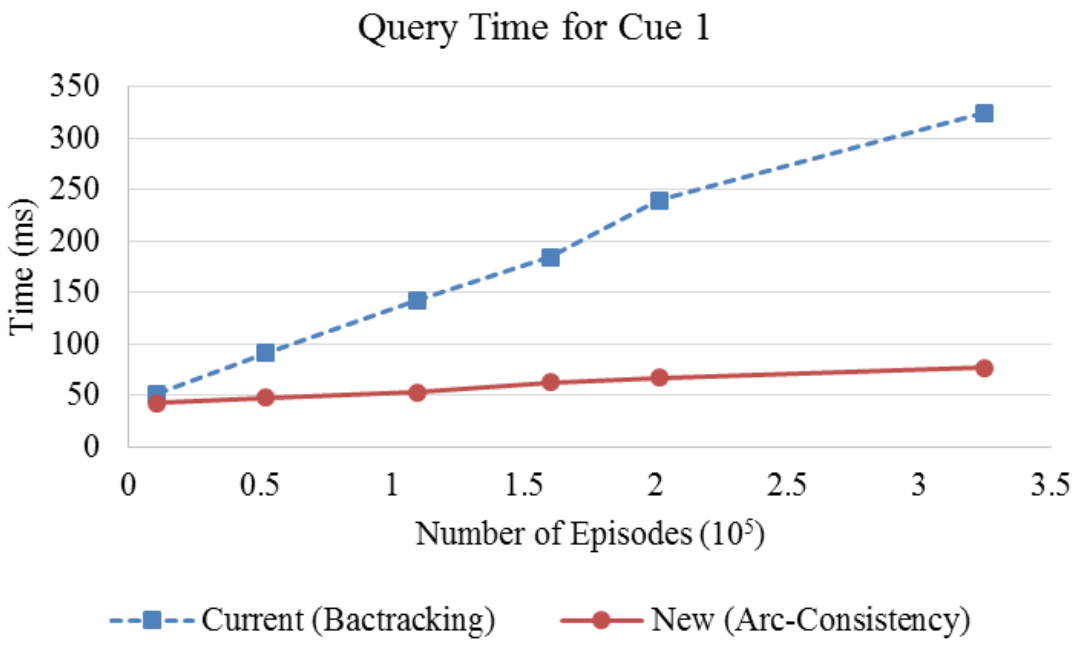


Figure 4-4: Query time of cue 1 from TankSoar episodic memory.

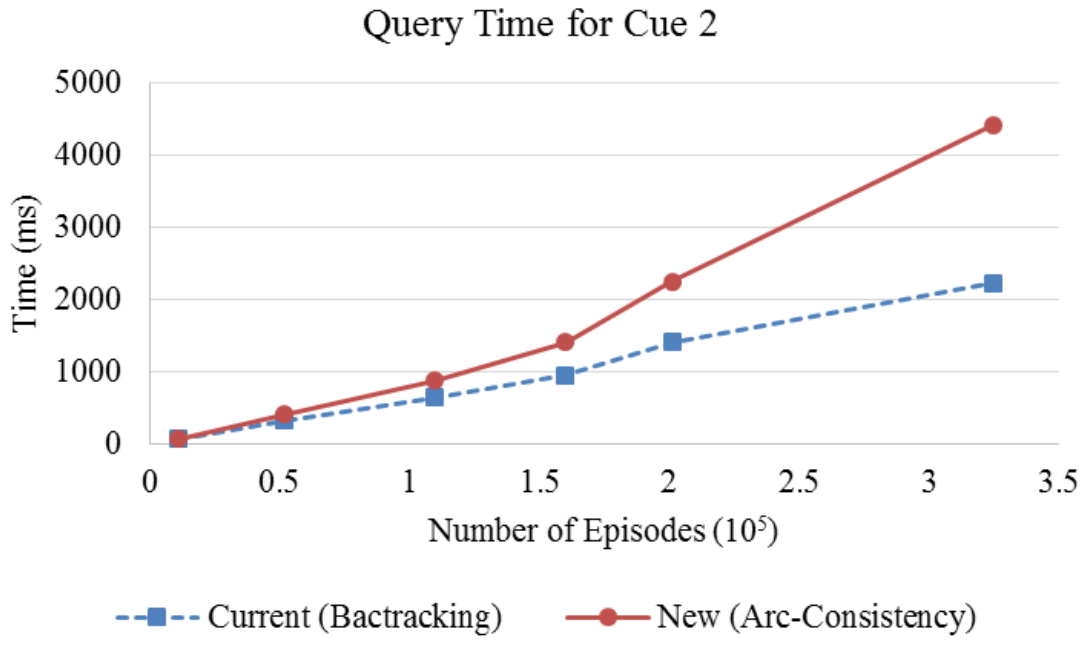


Figure 4-5: Query time of cue 2 from TankSoar episodic memory.

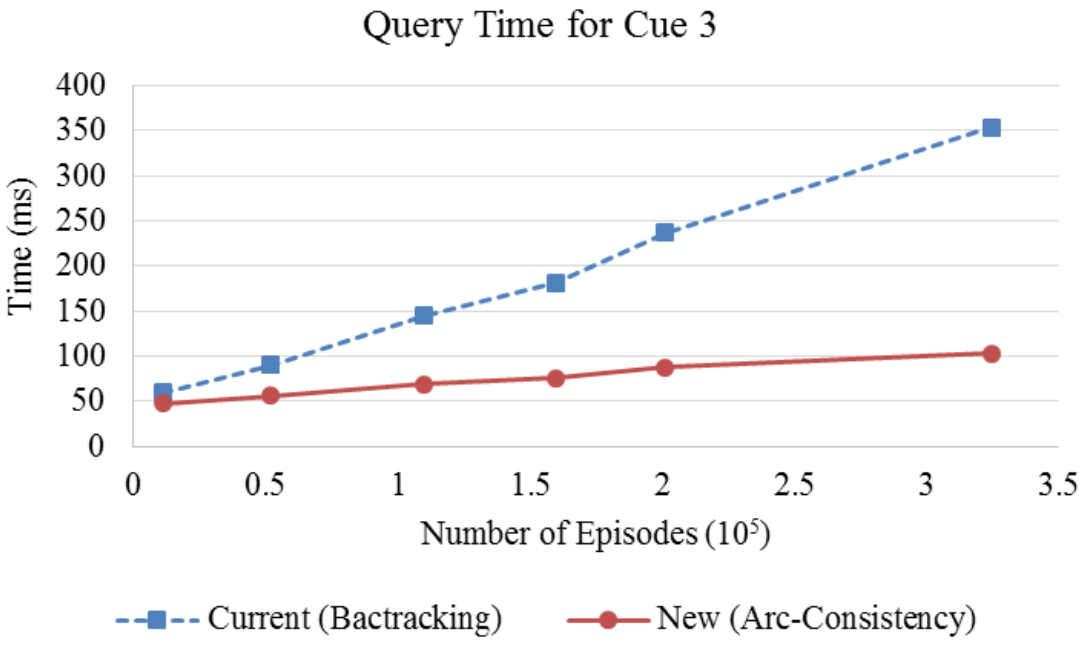


Figure 4-6: Query time of cue 3 from TankSoar Episodic Memory.

From the results obtained, it can be observed that the query time for both the algorithms gradually increases as the episodic memory size grows. However, the rate of increase in time is lower with our proposed algorithm. This is due to the multiple expensive graph match in most of the cases with BT, whereas arc-consistency tightens the constraints by assigning the domains to variables and removing multiple matches.

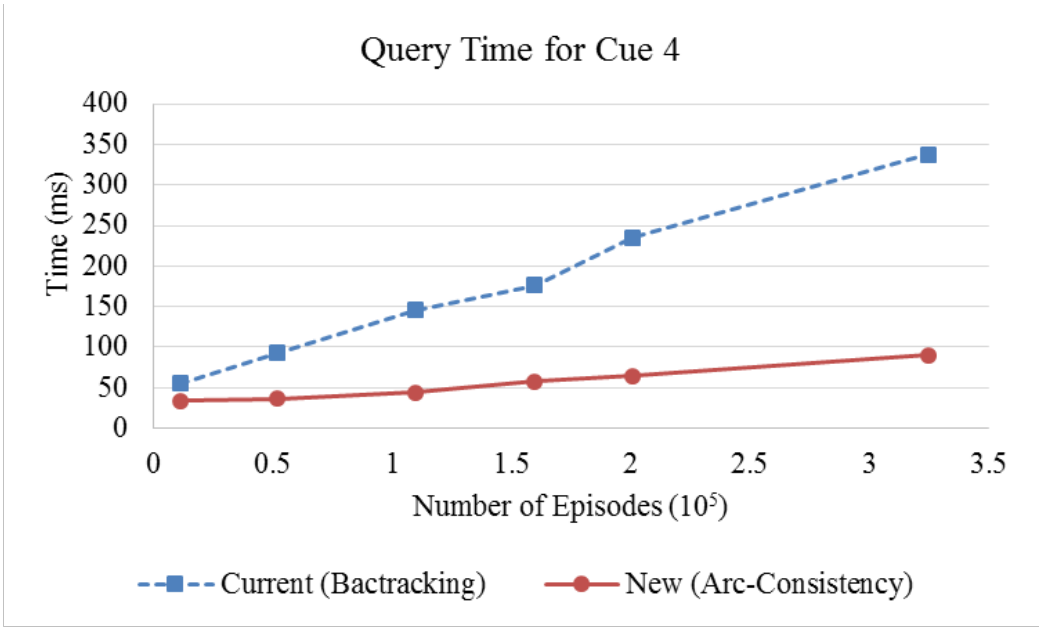


Figure 4-7: Query time of cue 4 from TankSoar episodic memory.

Fig. 4-5 shows that cue 2 requires large query time. This is because the number of matching candidates is large. In every rotation step, TankSoar radar is switched on. Moreover, the query time with BT is less than that of the proposed technique. In some cases, if graph match phase occurs early, Soar is able to complete the search early. Whereas in most of the cases, Soar implementation ended up having graph match multiple times.

The proposed algorithm assigns domain to variables by single value to make arc-consistent. This makes it backtrack free match and reduce the search time. The Soar implementation attempts expensive graph match many times in most of the cases.

However, in some cases, the graph match that Soar implements happens earlier in interval search. This results in quick completion of search in Soar. The proposed algorithm for structural match performs graph match once in each search. It is also able to reduce domain to single element each time to make arc-consistent.

4.6 Conclusion

In this chapter, the performance analysis of arc-consistency for structural graph matching to retrieve an episode from TankSoar environment is presented. This approach removes the redundant values from the domains, which reduces the search time. Arc-consistency also performs the graph match once in each search by reducing the domain to a single element each time to make arc-consistent. The approach taken in this thesis is able to reduce the time of querying as compared to that of original implementation of Soar architecture.

Soar-EpMem has effective encoding techniques of episodic memory as well as a cue matching approach. However, in some cases, the performance reduces due to frequent backtracking caused by perfect match, but no graph match occurs. This thesis proposes an arc-consistency as an alternative structural graph matching algorithm, which tightens constraints by assigning domains to variables in which the need of frequent backtracking is removed. This allows the new algorithm to perform better.

Hence, an episode retrieval time can be reduced through the implementation of arc-consistency as CSP solver for an efficient structural graph matching algorithm.

Chapter 5

Conclusive Remarks

The ultimate goal of this research is to improve the performance of retrieval from episodic memory in Soar through the incorporation of new techniques in two areas of cue matching steps. The techniques proposed in chapters 3 and 4 serve as ways to meet the objective of this research. Given that an episodic memory has been widely used in intelligent systems, as run-time of system increases, the size of memory becomes large, and retrieval time increases. A timely retrieval is required for the system to have effective use of episodic memory.

The primary idea behind the proposed scheme is to introduce approximation to lower the computational burden of working memory. Following that, implementation of an arc-consistency algorithm is used to reduce the time of matching by removing multiple backtracking.

Chapter 3 explains approximations of BLA and presents Eaters performance, applying different WMAs. The results prove that these approximations do not degrade the overall performance of the agents by a great factor, and large efficiency in computational time can be achieved as well. As the activation calculation is a part of the cue matching step, approximations can be implemented to reduce the episode retrieval time.

In chapter 4, a detailed work related to structural graph match as CSP and its so-

lution is discussed. The proposed algorithm is presented and tested with the episodic memory generated with real testing environment, i.e. TankSoar. Also, from the results obtained, it can be concluded that the query time using arc-consistency is lower in comparison to Soar implementation and the rate of increase of retrieval time is minimized with respect to episodic memory size for most of the cues provided.

Hence, with implementation presented in this thesis, for the cue matching of episodic retrieval, the episodic memory retrieval can be improved as the size of episodic memory becomes large during a long run in cognitive systems.

5.1 Future Work

While this thesis demonstrated the possibility of reduction of retrieval time of episodic memory, this section presents potential to have some of the probable future directions this research can take.

In addition, research can take more details on approximation of activation in working memory, which are needed to be studied with respect to working memory size and management of working memory. Another future direction for this thesis is applying the proposed algorithm of structural graph match, to be implemented in Soar architecture and check the performance of an agent.

5.1.1 Improvement in Computational Model

The agent behavior based on the approximations of working memory activation has been studied. Comparable results on performance are achieved. A detailed study of working memory size and the management of working memory is to be examined as a future goal. Moreover, the implementation of other computational models such as biological based or abstract based models can be studied to enhance episodic learning.

5.1.2 Enhancement of Structural Graph Match

The implementation of arc-consistency for structural graph match reduces the query time for most of the cases. The result produced has unusual cases with cues having a large number of matching episodes. In the future, other CSP solvers that best suit all types of cues needs to be studied.

5.1.3 More Complex Environment and Tasks

Additional domains and different sets of test cases for episodic memory also need to be examined. TankSoar and Eaters are artificial computer games useful for initial evaluation. However, exploring much longer runs for a larger episodic memory size with much complex cases is of high importance. In addition, evaluating the performance in real world environments is also a necessary action future research should consider.

References

[1] E. Tulving *et al.*, "Episodic and semantic memory," *Organization of memory*, vol. 1, pp. 381–403, 1972.

[2] J. Anderson and C. Lebiere, "The atomic components of thought lawrence erlbaum," *Mathway, NJ*, 1998.

[3] J. R. Anderson, *How can the human mind occur in the physical universe?* Oxford University Press, 2009.

[4] P. Langley, "Cognitive architectures and general intelligent systems," *AI magazine*, vol. 27, no. 2, p. 33, 2006.

[5] R. Sun, "The clarion cognitive architecture: Extending cognitive modeling to social simulation," *Cognition and multi-agent interaction*, pp. 79–99, 2006.

[6] D. E. Kieras and D. E. Meyer, "An overview of the epic architecture for cognition and performance with application to human-computer interaction," *Human-computer interaction*, vol. 12, no. 4, pp. 391–438, 1997.

[7] A. M. Nuxoll and J. E. Laird, "Extending cognitive architecture with episodic memory," *Ann Arbor*, vol. 1001, pp. 48 109–2121, 2007.

[8] N. Derbinsky and J. E. Laird, "Efficiently implementing episodic memory," in *International Conference on Case-Based Reasoning*. Springer, 2009, pp. 403–417.

[9] A. Newell, "The william james lectures, 1987. unified theories of cognition," 1990.

[10] J. E. Laird, *The Soar cognitive architecture*. MIT press, 2012.

[11] D. G. Tecuci and B. W. Porter, *A generic memory module for events*, 2007, vol. 68, no. 09.

[12] G. Gillund and R. M. Shiffrin, "A retrieval model for both recognition and recall." *Psychological review*, vol. 91, no. 1, p. 1, 1984.

[13] J. R. Anderson and G. H. Bower, "Recognition and retrieval processes in free recall." *Psychological review*, vol. 79, no. 2, p. 97, 1972.

[14] E. Tsang, "Foundations of constraint satisfaction," 1995.

[15] V. Kumar, "Algorithms for constraint-satisfaction problems: A survey," *AI magazine*, vol. 13, no. 1, p. 32, 1992.

[16] R. Dechter, *Constraint processing*. Morgan Kaufmann, 2003.

[17] K. D. Forbus, D. Gentner, and K. Law, "Mac/fac: A model of similarity-based retrieval," *Cognitive science*, vol. 19, no. 2, pp. 141–205, 1995.

[18] M. Lenz and H.-D. Burkhard, "Case retrieval nets: Basic ideas and extensions," in *Annual conference on artificial intelligence*. Springer, 1996, pp. 227–239.

[19] S. Wess, K.-D. Althoff, and G. Derwand, "Using kd trees to improve the retrieval step in case-based reasoning," in *European Workshop on Case-Based Reasoning*. Springer, 1993, pp. 167–181.

[20] R. H. Stottler, A. L. Henke, and J. A. King, "Rapid retrieval algorithms for case-based reasoning." in *IJCAI*, vol. 89. Citeseer, 1989, pp. 233–237.

[21] A. G. Francis and A. Ram, "The utility problem in case-based reasoning," in *Case-Based Reasoning: Papers from the 1993 Workshop*, 1993, pp. 160–161.

[22] S. Fox, D. B. Leake *et al.*, "Using introspective reasoning to refine indexing," in *IJCAI*, 1995, pp. 391–399.

[23] D. R. Wilson and T. R. Martinez, "Reduction techniques for instance-based learning algorithms," *Machine learning*, vol. 38, no. 3, pp. 257–286, 2000.

[24] D. W. Patterson, N. Rooney, and M. Galushka, "Efficient retrieval for case-based reasoning." in *FLAIRS Conference*, 2003, pp. 144–149.

[25] D. W. Patterson, M. Galushka, and N. Rooney, "An effective indexing and retrieval approach for temporal cases." in *FLAIRS Conference*, 2004, pp. 190–195.

[26] M. D. Jære, A. Aamodt, and P. Skalle, "Representing temporal knowledge for case-based prediction," in *European Conference on Case-Based Reasoning*. Springer, 2002, pp. 174–188.

[27] W. Wang, B. Subagdja, A.-H. Tan, and J. A. Starzyk, "A self-organizing approach to episodic memory modeling," in *Neural Networks (IJCNN), The 2010 International Joint Conference on*. IEEE, 2010, pp. 1–8.

[28] D. Wang, A.-H. Tan, and C. Miao, "Modeling autobiographical memory in human-like autonomous agents," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2016, pp. 845–853.

[29] W. Wang, B. Subagdja, A.-H. Tan, and J. A. Starzyk, "Neural modeling of episodic memory: Encoding, retrieval, and forgetting," *IEEE transactions on neural networks and learning systems*, vol. 23, no. 10, pp. 1574–1586, 2012.

[30] L. Shastri, "Episodic memory and cortico–hippocampal interactions," *Trends in cognitive sciences*, vol. 6, no. 4, pp. 162–168, 2002.

[31] A. Nuxoll, "Enhancing intelligent agents with episodic memory," University of Michigan. [Online]. Available: https://deepblue.lib.umich.edu/bitstream/handle/2027.42/57720/anuxoll_1.pdf?sequence=2&isAllowed=y

[32] B. V. Dasarathy, "Nearest neighbor ({NN}) norms:{NN} pattern classification techniques," 1991.

[33] G. M. Landau and U. Vishkin, "Fast parallel and serial approximate string matching," *Journal of algorithms*, vol. 10, no. 2, pp. 157–169, 1989.

[34] W. A. Burkhard, "Hashing and trie algorithms for partial match retrieval," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 2, pp. 175–187, 1976.

[35] J. W. Grzymala-Busse and A. Y. Wang, "Modified algorithms lem1 and lem2 for rule induction from data with missing attribute values," in *Proc. of the Fifth International Workshop on Rough Sets and Soft Computing (RSSC'97) at the Third Joint Conference on Information Sciences (JCIS'97), Research Triangle Park, NC*, 1997, pp. 69–72.

[36] J. Hakkinen and J. Tian, "N-gram and decision tree based language identification for written words," in *Automatic Speech Recognition and Understanding, 2001. ASRU'01. IEEE Workshop on.* IEEE, 2001, pp. 335–338.

[37] G. Hua and A. Akbarzadeh, "A robust elastic and partial matching metric for face recognition," in *Computer Vision, 2009 IEEE 12th International Conference on.* IEEE, 2009, pp. 2082–2089.

[38] N. Cercone, A. An, and C. Chan, "Rule-induction and case-based reasoning: Hybrid architectures appear advantageous," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 166–174, 1999.

[39] A. K. Mackworth, "Consistency in networks of relations," in *Readings in Artificial Intelligence*. Elsevier, 1981, pp. 69–78.

[40] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems," *Artificial Intelligence*, vol. 58, no. 1-3, pp. 161–205, 1992.

[41] J. Boutheina and K. Ghédira, "On the enhancement of the informed backtracking algorithm," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2003, pp. 967–967.

[42] J. Gaschnig, "A general backtrack algorithm that eliminates most redundant tests." in *IJCAI*, 1977, p. 457.

[43] K. Ghédira, *Constraint satisfaction problems: csp formalisms and techniques*. John Wiley & Sons, 2013.

[44] C. Bessiere, K. Stergiou, and T. Walsh, "Domain filtering consistencies for non-binary constraints," *Artificial Intelligence*, vol. 172, no. 6-7, pp. 800–822, 2008.

[45] R. Mohr and T. C. Henderson, "Arc and path consistency revisited," *Artificial intelligence*, vol. 28, no. 2, pp. 225–233, 1986.

[46] K. Mizuno, Y. Fukui, and S. Nishihara, "Urban traffic signal control based on distributed constraint satisfaction," in *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*. IEEE, 2008, pp. 65–65.

[47] P.-A. Yvars, "Using constraint satisfaction for designing mechanical systems," *International Journal on Interactive Design and Manufacturing (IJIDeM)*, vol. 2, no. 3, pp. 161–167, 2008.

[48] A. Vera, A. Howes, M. McCurdy, and R. L. Lewis, "A constraint satisfaction approach to predicting skilled interactive cognition," in *Proceedings of the SIGCHI conference on Human factors in computing systems.* ACM, 2004, pp. 121–128.

[49] S. Benbernou and M.-S. Hacid, "Resolution and constraint propagation for semantic web services discovery," *Distributed and Parallel Databases*, vol. 18, no. 1, pp. 65–81, 2005.

[50] G. Bella, S. Bistarelli, and S. N. Foley, "Soft constraints for security," *Electronic Notes in Theoretical Computer Science*, vol. 142, pp. 11–29, 2006.

[51] B. Faltings, T. Léauté, and A. Petcu, "Privacy guarantees through distributed constraint satisfaction," in *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, vol. 2. IEEE, 2008, pp. 350–358.

[52] I. Crabtree, "Resource scheduling: comparing simulated annealing with constraint programming," *BT Technology Journal*, vol. 13, no. 1, pp. 121–127, 1995.

[53] S. Lawrence, "Resource constrained scheduling: An experimental investigation of heuristic scheduling techniques. graduate school of industrial administration," 1984.

[54] N. Guerinik and M. Van Caneghem, "Solving crew scheduling problems by constraint programming," in *International Conference on Principles and Practice of Constraint Programming.* Springer, 1995, pp. 481–498.

[55] J. David and T. Chew, "Constraint-based applications in production planning: examples from the automotive industry," *Proceedings of Practical Applications of Constraint Technology (PACT'95). Practical Applications Company, Blackpool, UK*, pp. 37–51, 1995.

[56] P. Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in *International conference on principles and practice of constraint programming*. Springer, 1998, pp. 417–431.

[57] M. Dincbas, H. Simonis, and P. Van Hentenryck, "Solving the car-sequencing problem in constraint logic programming." in *ECAI*, vol. 88, 1988, pp. 290–295.

[58] U. Montanari, "Networks of constraints: Fundamental properties and applications to picture processing," *Information sciences*, vol. 7, pp. 95–132, 1974.

[59] P. Blache, "Constraints, linguistic theories, and natural language processing," in *International Conference on Natural Language Processing*. Springer, 2000, pp. 221–232.

[60] P. Van Beek, "Reasoning about qualitative temporal information," *Artificial intelligence*, vol. 58, no. 1-3, pp. 297–326, 1992.

[61] P. Moore, M. Jackson, and B. Hu, "Constraint satisfaction in intelligent context-aware systems," in *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*. IEEE, 2010, pp. 75–80.

[62] P. G. Kolaitis and M. Y. Vardi, "Conjunctive-query containment and constraint satisfaction," *Journal of Computer and System Sciences*, vol. 61, no. 2, pp. 302–332, 2000.

[63] A. A. Petrov, "Computationally efficient approximation of the base-level learning equation in act-r," in *Proceedings of the seventh international conference on cognitive modeling*. Citeseer, 2006, pp. 391–392.

[64] R. Chong, "The addition of an activation and decay mechanism to the soar architecture," in *Proc. of the 5th Intl. Conf. on Cognitive Modeling*, 2003, pp. 45–50.

[65] A. Nuxoll and J. E. Laird, "A cognitive model of episodic memory integrated with a general cognitive architecture." in *ICCM*. Citeseer, 2004, pp. 220–225.

Appendix A

CSoar Program for Eaters Agent

```
epmem --set learning on
epmem --set trigger dc
epmem --set balance 0
epmem --set graph-match off
epmem --set database file
epmem --set path firstBLA.db
epmem --set append on

wma --set activation on

################## Initialization operator
sp {propose*initialize-eater
   (state <s> ^superstate nil
              -^name)
-->
   (<s> ^operator <o> +)
   (<o> ^name eater)
   (cmd clog |foo.log|)
```

```
        (cmd clog -c)
}
sp {apply*initialize-eater
   (state <s> ^operator <op>)
   (<op> ^name eater)
-->
   (<s> ^name eater
                ^count 0)
}
# cleans the output-link once commands complete
sp {apply*cleanup*output-link
   (state <s> ^operator <op>
              ^superstate nil
              ^io.output-link <out>)
   (<out> ^<cmd> <id>)
#   (<op> ^name move)
   (<id> ^status)
-->
   (<out> ^<cmd> <id> -)
}
############################ misc useful elaboration rules
sp {elaborate*state*name
   (state <s> ^superstate.operator.name <name>)
-->
   (<s> ^name <name>)
}
```

```
sp {elaborate*state*top-state
   (state <s> ^superstate.top-state <ts>)
-->
   (<s> ^top-state <ts>)
}

sp {elaborate*top-state*top-state
   (state <s> ^superstate nil)
-->
   (<s> ^top-state <s>)
}

sp {elaborate*state*io
   (state <s> ^superstate.io <io>)
-->
   (<s> ^io <io>)
}

##########################################
sp {propose*move
   (state <s> ^name eater
                        ^io.input-link.my-location.
                                    <dir>.content
              { <content> <> wall })
-->
```

```
   (<s> ^operator <o> +)
   (<o> ^name move
        ^direction <dir>
        ^content <content>)
        (write | | <dir>)
        (write | | <content> (crlf))

}

sp {eater*elaborate*move-tie
   (state <s> ^impasse tie
              ^attribute operator
              ^choices multiple
                        ^superstate.name eater)
-->
   (<s> ^name move-tie)
   (write | "elaborate*move tie" | (crlf))
}

sp {propose*move-tie*evaluate
   (state <s> ^name move-tie
              ^item <ss-op>
             -^evaluated.name <dir>)
   (<ss-op> ^direction <dir>)
-->
   (<s> ^operator <o> + =)
```

```
    (<o> ^name evaluate
         ^direction <dir>)
        (write | | <dir> (crlf))
        (write | "propose*move-tie*evaluate" |(crlf))
}
############################################################
sp {propose*evaluate*query
    (state <s> ^name evaluate
               -^epmem.command.query
               -^epmem.command.next)
-->
    (<s> ^operator <o> + =)
    (<o> ^name query)
    (write | "propose*evaluate*query" | (crlf))
}

sp {apply*query
    (state <s> ^operator <o>
                ^epmem.command <cmd>
                ^superstate.operator.direction <dir1>)
        (<o> ^name query)
-->
    (<cmd> ^query <q>)
    (<q> ^io <io>)
    (<io> ^output-link.move.direction <dir1>)
    (<io> ^input-link.foo <bar>)
```

```
}
#############################################################
sp {propose*evaluate*result
   (state <s> ^name evaluate
              ^epmem.command.query
              ^epmem.result.<< success failure >>)
-->
   (<s> ^operator <op> + =)
   (<op> ^name result)
}

sp {apply*result*failure
   (state <s> ^operator <op>
              ^epmem <epmem>
              ^superstate <ss>
              ^quiescence t)
   (<ss> ^operator <ss-op>)
   (<op> ^name result)
   (<epmem> ^result.failure
            ^command.query.io.output-link.
                        move.direction <dir>)
-->
   (<ss> ^evaluated <e>)
   (<e> ^name <dir>
        ^found f
        ^value 0)
```

```
        (write | | failure)
        (write | | <dir> (crlf))
}
############################################################
sp {apply*result*success
   (state <s> ^operator <op>
                          ^epmem <epmem>
              ^epmem.command <cmd>
                          ^superstate <ss>)
   (<epmem> ^result.retrieved.io.input-link.eater.score <v1>)
   (<cmd> ^query <q>)
-->
   (<cmd> ^query <q> -
          ^next <n>)
        (<s> ^first-score <v1>)
        (write | | first-score (crlf))
}
sp {propose*evaluate*result2
   (state <s> ^name evaluate
              ^epmem.command.next
              ^epmem.result.<< success failure >>)
-->
   (<s> ^operator <op> + =)
   (<op> ^name result2)
}
sp {apply*result2*success
```

```
   (state <s> ^operator <op>
              ^epmem <epmem>
              ^superstate <ss>
                          ^first-score <f1>)
   (<ss> ^operator.direction <dir>)
   (<op> ^name result2)
   (<epmem> ^result.retrieved.io.input-link.eater.score <v>)
-->
   (<ss> ^evaluated <e>)
   (<e> ^name <dir>
        ^found t
        ^value (+ (- <f1>) <v>))
        (write (crlf) | | success)
        (write (crlf) | | <v>)
        (write (crlf) | | (- <f1>))
}
#########################################################################

sp {propose*move-tie*copy
   (state <s> ^name move-tie)
-->
   (<s> ^operator <op> + <)
   (<op> ^name copy)
}
sp {apply*copy
   (state <s> ^operator <op>
```

```
                ^superstate <ss>
                ^item <ss-op>
                ^evaluated <e>)
    (<op> ^name copy)
    (<ss-op> ^direction <dir>)
    (<e> ^name <dir>
         ^value <v>)
-->
    (<ss> ^operator <ss-op> = <v>)
     (write (crlf)| | <dir>)
         (write | | <v> (crlf))
}
##########################################

sp {apply*move
    (state <s> ^io.output-link <ol>
                ^operator <o>
                            ^io.input-link.eater.score <score>)


    (<o> ^name move
         ^direction <dir>)
-->
    (<s> ^operator <o> + =)
    (<o> ^name remove)
    (<ol> ^move.direction <dir>)
```

```
    (write (crlf)|move direction:| <dir> (crlf))
   (write (crlf)|score:| <score> (crlf))
}

# Apply*move*remove-move:
# If the move operator is selected,
# and there is a completed move command on the output link,
# then remove that command.

sp {apply*move*remove-move
    (state <s> ^io.output-link <ol>
               ^operator.name remove)

    (<ol> ^move <mv>)
    (<mv> ^status complete)
    (<ol> ^<cmd> <id>)
-->
         (<ol> ^move <mv> -)
         (write | "move removed"| (crlf))
}
```

Appendix B

Episodic Memory: Evaluation Cues for TankSoar

Cue 1.
```
(<cmd> ^query <cue>)
(<cue> ^io <io>)
(<io> ^input-link <il>)
(<il> ^x 1)
(<il> ^y 1)
```

Cue 2.
```
(<cmd> ^query <cue>)
(<cue> ^io <io>)
(<io> ^output-link <ol>)
(<ol> ^radar <sw>)
(<sw> ^switch on)
```

Cue 3.
```
(<cmd> ^query <cue>)
(<cue> ^io <io>)
(<io> ^input-link <il>)
(<io> ^output-link <ol>)
```

```
(<ol> ^rotate <dir>)
(<dir> ^direction left)
(<il> ^health 1000)
(<il> ^energy 1000)
```

Cue 4.

```
(<cmd> ^query <cue>)
(<cue> ^io <io>)
(<io> ^input-link <il>)
(<io> ^output-link <ol>)
(<ol> ^rotate <dir>)
(<dir> ^direction left)
(<ol> ^radar-power <set>)
(<set> ^setting 13)
(<il> ^blocked <b>)
(<b> ^left no)
```

Appendix C

Eaters Agent Simulation Results

Table C.1: Comparison of three activation functions using average score of Eaters agent after 100 actions.

Activation Functions	**Iteration 1**	**Iteration 2**	**Iteration 3**
Base-level	371	458	506
Petrov's Approximation	349	411	445
Improved Approximation	370	421	491

Appendix D

TankSoar Episodic Memory Log and Query Time For Retrieval

Table D.1: TankSoar episodic memory log with number of episodes.

S.N.	Number of Episodes	Tag
1	11142	Ep1
2	51807	Ep2
3	109793	Ep3
4	160087	Ep4
5	201116	Ep5
6	324771	Ep6

Table D.2: Total query time of four different cues for TankSoar episodic memory.

Tag	Query Time for Different Cues							
	Cue 1		Cue 2		Cue 3		Cue 4	
	BT	AC	BT	AC	BT	AC	BT	AC
Ep1	51.752	42.328	63.541	69.342	59.5824	47.3486	54.8424	33.834
Ep2	90.964	47.828	319.051	411.434	90.3942	56.0046	92.7468	36.508
Ep3	142.2986	52.85	638.789	874.82	144.49	68.638	145.554	43.728
Ep4	184.0808	62.822	956.761	1413.83	181.327	75.664	175.907	57.784
Ep5	239.5742	67.542	1410.44	2251.3	327.123	87.348	235.161	65.01
Ep6	324.5454	76.998	2236.22	4432.42	353.916	103.42	337.296	90.196