An Episodic Memory Retrieval Algorithm for the Soar Cognitive Architecture

Francis Li, Jesse Frost, Braden J. Phillips

School of Electrical and Electronic Engineering, University of Adelaide
{francis.li,jesse.frost,braden.phillips}@adelaide.edu.au

Abstract. Episodic memory in cognitive architectures allows intelligent agents to query their past experiences to influence decision making. Episodic memory in the Soar cognitive architecture must efficiently encode, store, search and reconstruct episodes as snapshots of short term working memory. The performance of the current search algorithm can be improved by doing a structural match phase first on all stored data and enforcing arc consistency in the first stage of the structural match. We demonstrate experimentally the performance of the improved search algorithm in two Soar environments.

1 Introduction

1.1 The Soar Cognitive Architecture

The Soar cognitive architecture is a model of cognition based on a symbolic production rule system. Soar models problem solving by moving an agent through the problem state-space. At each step it uses a decision making algorithm to decide how to modify the problem state [5].

Soar internally models the problem state in temporary memory called *Working Memory* (WM), which consists of a set of 3-tuples called *Working Memory Elements* (WMEs). Each WME consists of symbols which are respectively named the identifier, attribute and value. Together, all WMEs in WM represent a connected, directed, rooted graph where:

- identifiers represent inner nodes;
- attributes represent edge labels; and
- values represent leaf nodes or inner nodes.

Figure 1 shows an example of how WMEs are mapped to the graph representation in Soar. The state of an agent's WM represents its current knowledge of the problem state. The agent moves through the problem state-space by adding, removing and changing WMEs. In this way, the graph structure represented by WM changes over time.

```
(S1 ^thing A1)
(S1 ^thing B1)
(A1 ^value a)
(A1 ^value b)
(B1 ^value c)
```

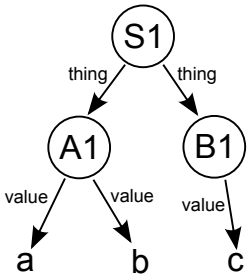


Fig. 1. A set of WMEs on the left and the associated graph representation on the right. In this example `S1`, `A1` and `B1` are *identifiers*, `thing` and `value` are *attributes* and `a`, `b` and `c` are *values*.

1.2 Soar's Episodic Memory Implementation

In addition to WM, Soar can make use of *Episodic Memory* (EpMem) to assist its decision making process. EpMem automatically stores the state of WM at a specified phase in each decision cycle. It maintains a complete history of the state of WM throughout the agent's existence as a set of snapshots of WM called *episodes*.

During any cycle the agent can *query* EpMem to find past experiences similar to its current state and use past decisions to influence its decision process in the present. It does this by providing EpMem with a cue — a data structure representing the pattern to search for [4]. EpMem searches its history and returns the most recent episode containing an instance of the cue structure [6]. Figure 2 illustrates the relationship between Soar's WM and EpMem.

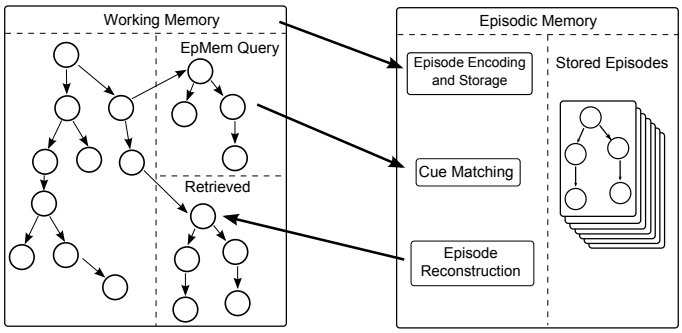


Fig. 2. The relationship between EpMem and WM in Soar. Episodes are stored in EpMem as encoded representations of WM and are retrieved by searching and matching against a cue.

A complete detailed description of the current implementation of EpMem in Soar can be found in [3]. An EpMem implementation must be able to:

– Efficiently encode, store and index a large set of slowly changing episodes.
– Rapidly find the episode that best matches a cue according to some 'best match' criteria.
– Reconstruct/decode the matching episode and return it in the agent's WM.

Encoding and storage. Soar's current implementation of EpMem exploits the temporal redundancy of episodes — since WM changes slowly, episodes that are close in time share many common WMEs. Episodes are encoded by separating structural and temporal information. The history of all structures that have appeared in WM is stored in the *Working Memory Graph* (WMG) — an encoding of every WME that has ever appeared in any episode[1]. Each entry in the working memory graph contains an index to a table of *intervals*. This table encodes the episodes for which each structure in the WMG was present in WM. The size of the episode store is linear only in the *changes* to WM, as each addition or deletion of a WME between episodes creates a new interval entry.

Cue matching. A cue is a set of WMEs which together represent a rooted, directed, acyclic *subgraph* of WM. A match for a cue is defined as the most recent episode that shares the greatest number of features in common with the cue leaf nodes. The cue matching algorithm uses a two phase process to avoid performing a complete graph match for every stored episode: an episode is only a candidate for for the graph match if it contains all *surface* features of the cue, where a surface feature is a unique directed path from root to leaf. The matching algorithm searches for candidate episodes by 'walking' backwards through the interval table. Each interval represents an episode where the state of WM changed. For each change, the state of satisfaction of the surface features is updated by propagating changes through a *DNF graph* [2] compiled from the cue.

A perfect surface match does not guarantee that the cue structure was found in a past episode, only that all surface features were present. A structural graph match step is used to determine whether the cue structure is present within the candidate episode using a simple backtracking algorithm. If this search succeeds, the episode is returned, otherwise the interval walk continues. If the interval walk exhausts the episode store, the most recent episode with the highest surface match score is returned.

Reconstruction. The reconstruction algorithm takes an episode number from the cue matching algorithm and returns the full episode to the agent in WM. Further details of episode reconstruction are outside the scope of this discussion.

1.3 Contributions

Searching EpMem is the NP-complete *subgraph isomorphism* problem. Soar reduces the search cost by implementing a two-phase matching algorithm. This

[1] EpMem does not actually store certain working memory structures, such as those related to the EpMem interface itself.

does an inexpensive *surface match* to filter the potential episodes before they are subjected to an expensive *structural graph match* (Section 1.2). Although this algorithm is effective, there are certain WM and cue structures for which retrievals perform poorly. We address these issues by making the following contributions:

- We describe the structural match as a constraint satisfaction problem (CSP) (Section 2.1). Soar's structural match has not been extensively studied [3], so formalising the search under a known framework allows us to leverage well-known techniques to improve search time cost. We enforce arc consistency to tighten the constraints before backtracking, thus reducing the amount of dead-ends hit during search.
- We perform the structural match before checking that the temporal information is consistent (Section 2.1, 2.2). This allows us to do the most constrained search first, and ensures that the graph match is performed only once.
- We show that the proposed algorithm obtains comparable performance on most cues, and exceeds performance for cues where the original algorithm performs poorly (Section 3).

2 A New Episodic Memory Retrieval Algorithm

Soar's retrieval algorithm terminates quickly if a surface match is likely to correspond with a complete graph match. However:

- if the stored episodes contain many structures that return perfect surface matches, but fail the structural graph match, the graph match step will be executed many times.
- if the cue contains WMEs which have the same identifier and attribute but different values (multi-valued attributes), the backtracking step could search many dead-ends and backtrack frequently.

We propose an alternative search algorithm which:

- tightens constraints before backtracking to avoid the second slow retrieval scenario.
- performs the structural match once before doing the interval walk — this removes the possibility of repeating the graph match and avoids the first slow retrieval scenario.

2.1 Structural Match

Finding an instance of the cue structure in the WMG is a constraint satisfaction problem. A constraint satisfaction problem (CSP) is defined as a set of variables $\mathcal{X} = \{X_1, X_2, \ldots X_n\}$, a set of constraints $\mathcal{C} = \{C_1, C_2, \ldots C_m\}$ and a set of non-empty domains $\mathcal{D} = \{D_1, D_2, \ldots D_n\}$ [1].

- The variable $X_i \in X$ may take on a value from $D_i \in D$.

- The k-ary constraint C_i is a pair $\langle S_i, R_i \rangle$ where $S_i \subset \mathcal{X}$, called the scope of C_i, is a subset of k variables $(X_{i_1}, \ldots, X_{i_k})$ and $R_i \subset D_{i_1} \times \ldots \times D_{i_k}$ is a k-ary relation on S_i.
- A unary constraint is one for which $k = 1$ and a binary constraint is one for which $k = 2$.

An instantiation of the variables is an assignment of values to each variable from the corresponding domain. An instantiation satisfies a constraint C_i if it satisfies the relation R_i. An instantiation that satisfies all constraints is a solution to the CSP.

We can define an EpMem cue to be a set of n WMEs $\{W_1, \ldots W_n\}$. For each unique inner node label in this set, define a variable X_j and a domain D_j which is the set of all unique inner node labels stored in the WMG. For each $W_i \in \{W_1, \ldots W_n\}$, define a constraint C_i. If W_i has a single inner node label (i.e. if it represents an edge terminating on a leaf node), C_i is a unary constraint. If W_i has two inner node labels (i.e. if it represents an edge between two inner nodes), C_i is a binary constraint. A WME in the WMG satisfies the relation R_i if the corresponding edge labels and leaf nodes match the corresponding edge labels and leaf nodes in W_i.

Solutions to a CSP are generated through search. Simple backtracking is a depth first search where partial instantiations are extended by assigning values to variables and then ensuring that all constraints are still satisfied. If a constraint is violated, the search has reached a *dead-end* and must backtrack by changing the newly assigned variables value to another in its domain. A depth first search is $O(b^d)$, where b is the branching factor (the size of the domains) and d is the depth (the number of variables). We can do better than the worst case by preprocessing the problem to reduce the sizes of the domains.

A CSP is *arc-consistent* if any value in the domain of a variable can be extended by any other variable. Enforcing arc consistency shrinks the domains of all variables and makes search more efficient. The *revise* operation enforces *directional arc-consistency* (DAC) between two variables x_i and x_j between which there is a binary constraint C_ij by deleting the values in the domain D_i of x_i which are not in the relation R_ij. The complexity of revise is $O(b^2)$, as every value in D_i must be compared with every value in D_j. However, this is reduced to $O(b)$ in practice, as domains are stored in a hashed data structure.

A *constraint graph* is a graphical representation of a CSP, where variables are nodes, and constraints are edges between nodes. If a constraint graph is a tree under some ordering $d = (x_1, \ldots, x_n)$, then we can enforce DAC along that ordering d by ensuring that every variable x_i is arc-consistent relative to every variable x_j such that $i \leq j$ (see [1]). This performs the revise operation at most c times, where c is the number of binary constraints (i.e., the number of WMEs in the cue). So enforcing DAC has time complexity $O(cb)$.

The cue is rooted and acyclic and always reduces to a tree constraint graph. So the total cost of doing the structural match is the time taken to enforce DAC added to the time needed to generate all solutions through backtracking. The

backtracking stage is much more efficient as dead ends are never hit. All generated solutions are sent to the temporal match for the next phase of processing.

2.2 Temporal Match

Solutions from the structural graph match phase must be checked to ensure that each WME was present in WM at the *same* time — i.e. during a single episode. We compile the solutions into a data structure which tracks satisfaction of structural solutions as we walk through the intervals (in contrast to the DNF graph which tracks satisfaction of surface features). This data structure is effectively a conjunctive normal form (CNF) statement, comprised of a conjunction of clauses (which are the WMEs in the cue), where each clause is a disjunction of literals (possible values which WMEs may take on).

As we proceed with the interval walk, we keep track of the number of clauses which are 'on'. If this count reaches the maximum possible, then a solution is present in the same episode and we can return the id to the EpMem reconstruction algorithm. If not, we proceed until the interval walk exhausts all episodes, then return the episode with the maximum count.

The definition of a *partial match* has changed from the number of surface features to the number of structural solution WMEs present. It remains to be seen how this affects agents which rely on partial match in their decision cycles. However, this temporal match algorithm can be modified to support the traditional partial match criteria by altering the compilation step. There is further scope for study and optimisation of the temporal match component.

3 Experiments

We used two environments to evaluate EpMem algorithm performance:

- *TankSoar*[2] — a computer simulation which uses Soar agents to control virtual tanks which navigate a two dimensional map — which we chose as it was used in [3] as a test environment to evaluate the original EpMem algorithm.
- A simple implementation of the popular computer game *2048*[3] (where a player, in this case a Soar agent must move a set of tiles around on a four by four grid). We selected this as it produced WM structures similar to those that caused slower performance in the TankSoar environment.

The TankSoar agent *mapping-bot* receives sensor information on its input-link, and keeps track of the game's map internally. The 2048 environment models its game state as a four-by-four grid. Each cell in the grid has a column, row and value. The 2048 agent searches for past episodes which match some number of cells (see Figure 4).

We used the existing Soar EpMem search implementation and four different cues to search EpMem ranging in size from 1 to 320000 episodes generated from

[2] `http://soar.eecs.umich.edu/articles/downloads/domains/176-tanksoar`
[3] Original version available: `http://gabrielecirulli.github.io/2048/`

the TankSoar environment. These cues were taken from test data in [3] and can be seen in Figure 3. We then performed the test on the same EpMem data using the proposed algorithm implemented in the Java programming language.

We also used both search implementations to search for cues containing between 1 and 16 'cells' in episodic memory of up to 900 episodes generated by the 2048 environment. The structure of these cues is shown in Figure 4.

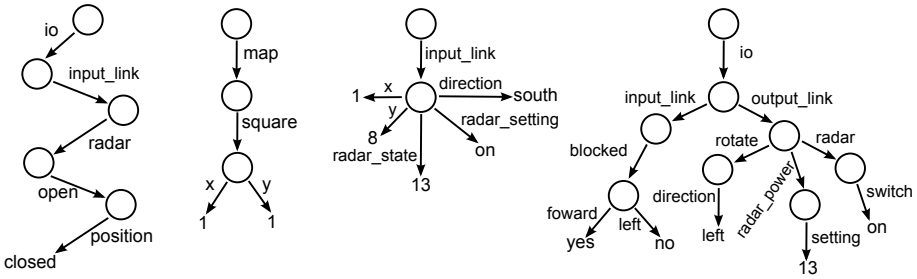


Fig. 3. From left to right, cue 1, cue 2, cue 3 and cue 4 used to search TankSoar's EpMem.

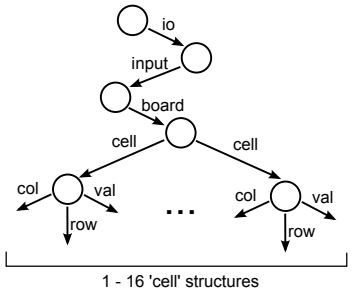


Fig. 4. The cue structure used by the 2048 agent contains 1-16 'cell' structures.

3.1 Results

The run time for both search implementations in the TankSoar environment is shown in Figure 5. Both were comparable in performance, with the proposed algorithm having a small advantage despite the addition of the preprocessing step. Figure 6 shows the runtime for a search in the 2048 episodic memory using both algorithms with cues containing between 1 and 4 cells. The proposed algorithm was consistently able to reduce the variable domains to a single element which resulted in backtrack free structural match. The Soar implementation consistently

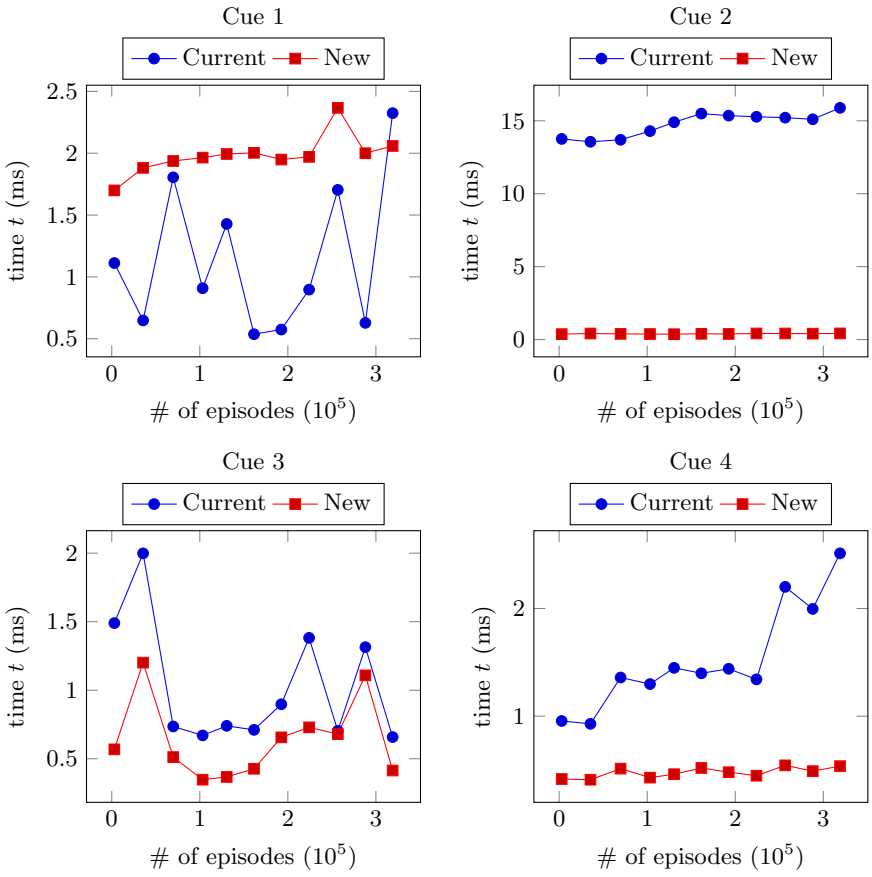
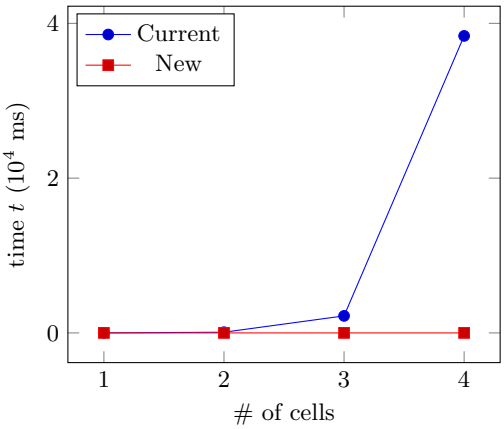


Fig. 5. The search time for the four TankSoar cues. Both algorithms are comparable except for cue 2, where the new algorithm is an order of magnitude better.

passed the surface match phase and triggered a graph match for many intervals. The unreduced domains caused near worst case time for each repeated attempt at structural graph match. We attempted to test cues with more than 4 cells, but tests required too much time to execute (greater than 30 minutes).

Figure 7 shows the search times for a cue consisting of two cells and an episodic memory of 1 to 900 episodes. The Soar implementation was in some cases able to complete the search early when the graph match phase matched early in the interval walk, but in other cases ended up attempting an expensive graph match many times. The proposed algorithm consistently performed better as it performed the graph match only once in each search and was able to reduce each domain to a single element each time.



	Time (ms)	
Cells	Current	New
1	3.29	0.31
2	90.5	0.4
3	2,200.13	0.53
4	38,375.94	0.69

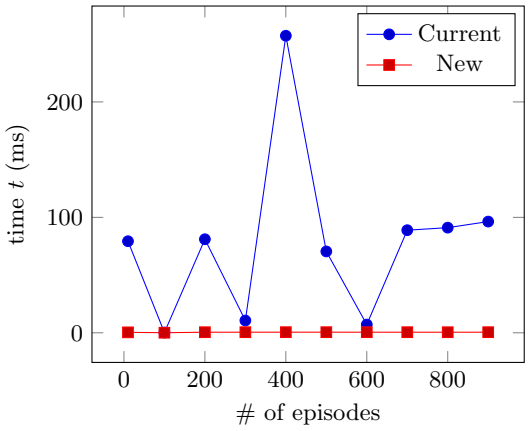
Fig. 6. Average search time for Soar increases exponentially as the number of cell structures in the cue are increased.

4 Conclusion

Soar's EpMem system has an effective encoding and cue matching algorithm. However, certain episode and cue structures can significantly reduce its performance by causing repeated attempts at performing a full graph match, or by stressing the simple backtracking algorithm.

We have proposed an alternative cue matching system that performs an efficient graph match first before attempting the interval search. The graph match enforces arc consistency before backtracking to tighten the constraints and avoid dead-ends. An implementation of this approach in the Java programming language provides comparable performance to the current algorithm in 'nice' cases, and significant improvements in the 'naughty' cases.

In addition, the proposed algorithm can extend EpMem to handle cyclic and non-rooted cues and has scope for further optimisation.



Episodes	Time (ms) Current	New
10	79.32	0.4
100	0.21	$6.39 \cdot 10^{-2}$
200	81.06	0.5
300	10.62	0.52
400	257.34	0.53
500	70.48	0.53
600	7.14	0.52
700	88.89	0.52
800	91.08	0.5
900	96.34	0.52

Fig. 7. Average search time for Soar vs. current implementation with 2-cell cue. Soar's performance is highly dependent on the number of times it needs to do a graph match.

References

1. Dechter, R.: Constraint processing. Morgan Kaufmann (2003)
2. Derbinsky, N., Laird, J.E.: Efficiently implementing episodic memory. In: Case-Based Reasoning Research and Development, pp. 403–417. Springer (2009)
3. Derbinsky, N.L.: Effective and efficient memory for generally intelligent agents. Ph.D. thesis, The University of Michigan (2012)
4. Laird, J.E.: Extending the soar cognitive architecture. Frontiers in Artificial Intelligence and Applications 171, 224 (2008)
5. Laird, J.E.: The Soar Cognitive Architecture. The MIT Press (2012)
6. Nuxoll, A., Laird, J.E.: A cognitive model of episodic memory integrated with a general cognitive architecture. In: ICCM. pp. 220–225. Citeseer (2004)