Building Behavior Trees from Observations in Real-Time Strategy Games

Glen Robertson
Department of Computer Science
University of Auckland
Auckland, New Zealand, 1010
Email: glen@cs.auckland.ac.nz

Ian Watson
Department of Computer Science
University of Auckland
Auckland, New Zealand, 1010
Email: ian@cs.auckland.ac.nz

***Abstract*—This paper presents a novel use of motif-finding techniques from computational biology to find recurring action sequences across many observations of expert humans carrying out a complex task. Information about recurring action sequences is used to produce a behavior tree without any additional domain information besides a simple similarity metric – no action models or reward functions are provided. This technique is applied to produce a behavior tree for strategic-level actions in the real-time strategy game StarCraft. The behavior tree was able to represent and summarise a large amount of information from the expert behavior examples much more compactly. The method could still be improved by discovering reactive actions present in the expert behavior and encoding these in the behavior tree.**

I. Introduction

An ongoing challenge in Artificial Intelligence (AI) is to create problem-solving agents that are able to carry out some task by selecting a series of appropriate actions to get from a starting state to achieve a goal – the field of planning. Ideally these agents would be able to be applied to the many practical problems that require a sequence of actions in order to carry out a task, such as robotic automation, game playing, and autonomous vehicles. However, applying a classical planning agent to a new domain typically requires significant knowledge engineering effort [1]. It would be preferable if domain knowledge could be learned automatically from examples, but current automated planning systems capable of learning domain knowledge are generally designed to operate under strong assumptions that do not hold in complex domains [2]. Conversely, case-based planning systems capable of acquiring domain knowledge can make few assumptions about the domain, but can have difficulty reacting to failures or exogenous events [3].

In many potential application areas, a planner capable of transitioning from any starting state to any goal state is not actually required, and instead it is sufficient or even desirable to have an agent capable of robustly carrying out a specific task or behavior. For example, in game playing, there is usually a very similar starting state and goal for each match or activity within a game – in board games this is the starting board layout and object of the game, and in video games this could be the starting and win conditions of a match or the daily activities of a non-player character. In the genre of real-time strategy games, video game industry developers tend to use scripting and finite state machines instead of more complex approaches because those techniques are well-understood, easy to customise, and are sufficient to produce the desired behavior [4], [5].

Since their introduction by the video game industry in 2005 [6], Behavior Trees (BTs) have become increasingly common in the industry for encoding agent behavior [4], [7], [8]. They have been used in major published games [6] and they are supported by major game engines such as Unity[1], Unreal Engine[2], and CryEngine[3]. BTs are hierarchical goal-oriented structures that appear somewhat similar to Hierarchical Task Networks (HTNs), but instead of being used to dynamically generate plans, BTs are static structures used to store and execute plans [4], [9]. This is a vital advantage for game designers because it allows them fine control over agent behavior by editing the BT, while still allowing complex behavior and behavior reuse through the hierarchical structure [4], [9]. Although they have fixed structure, BTs produce reactive behaviour by the interaction of conditional checks and success and failure propagation within the hierarchy. Various types of nodes (discussed further in section V) can be composed to produce parallel or sequential behavior, or choose amongst different possible behaviors based on the situation [9].

We are creating a system able to automatically learn domain knowledge from examples of expert behavior, with few assumptions about the domain, and be able to quickly react to changes in state during execution. This would combine some of the benefits of learning systems in automated planning and case-based planning. Instead of learning a set of planning operators, we aim to automatically learn to carry out a single complex task within a domain, creating a less-flexible but still widely applicable planning system. The learned knowledge will be represented and acted upon in the form of a BT, which is ideal for a single task within a domain. Furthermore, the resulting BT is able to be hand-customised, so this approach could be used as an initial step, followed by human refinement, in the process of defining new behavior for an agent.

In the remainder of this paper we start by outlining related work to automatically learning planning knowledge in the form of HTNs, case-based planners, and BTs. We concretely define the challenging problem of learning a task from observations

[1]Unity — Behavior Designer:
https://www.assetstore.unity3d.com/en/#!/content/15277

[2]Unreal Engine — Behavior Trees:
https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/

[3]CryEngine — Modular Behavior Tree:
http://docs.cryengine.com/display/SDKDOC4/Modular+Behavior+Tree

of expert behaviour, and outline the domain of the Real-Time Strategy (RTS) game StarCraft as our motivating example. We then present our approach to the first part of the learning system: using a motif-finding technique to find and collapse repeated patterns of actions. We present some results from the current system and discuss its limitations. Finally we discuss potential future directions and conclude the paper.

II. Related Work

Early automated planning systems such as STRIPS [10] made strong assumptions about the domain in order to operate, such as a fully observable, deterministic world that changes only due to agent actions, and actions that are sequential and instantaneous, with known preconditions and effects. More recent work has aimed to make planning more practically applicable by automatically learning action models [1], [11]–[13], task models [14], [15], or hierarchical structure [16], [17]. Some work also expands the applicability of planners by relaxing assumptions from classical planning, addressing learning with nondeterminism [18], [19], partial observability [13], [20]–[22], or durative actions [2]. Almost all of this work on learning in automated planning learns by observing plan executions, including observations of the world state, as carried out by an external expert, allowing the learner to get a good coverage of the common cases in what could be a huge (or infinite) space of possible actions and observations. All of this work still requires strong assumptions about the domain, or domain knowledge provided, usually in the form of accurate action models.

An alternative approach to learning from examples of expert behavior is case-based planning, which finds solution action sequences by retrieving and adapting previously-encountered solutions to similar problems. Unlike learning in automated planning, which focuses on acquiring logical building blocks for the planner, case-based planners learn associations between initial states and partial or complete plans as solutions. Case-based planners can operate with very little domain knowledge and few assumptions about the domain, but because they do most of the processing at runtime (for the retrieval and adaptation parts of the case-based planning process), they can have efficiency issues when problems have large case bases or time constraints [23]. Case-based planning also face difficulty in adapting solutions for particular circumstances – long solutions may react slowly to unexpected outcomes during execution, while short solutions may react excessively to small differences in state or have difficulty reasoning about action ordering [3], [24]. A possible remedy to these issues is to introduce conditional checks and hierarchical structure into cases [3], [23], [24].

Other work has examined building probabilistic behavior models using Hidden Markov Models [25] and Bayesian Models [26] from examples. These approaches require very little domain knowledge and are capable of recognising or predicting plans. However, they are not designed to be used for creating plans – their predictions could be extrapolated into a plan but this would likely lead to increasing error and cyclic behavior. There are also task-learning methods based on explanation based learning [15], in which the agent explores the domain but also interacts with a human teacher in order to learn. This requires an action model for the exploration phase, and a human operator with domain knowledge in the interaction phase.

Instead of focusing on learning from examples, some work has instead used genetic algorithms to evolve BTs in an exploratory process [27], [28]. These approaches hold promise but can become prohibitively computationally expensive for complex domains. They also require the addition of a fitness function for evaluating evolved BTs. To the best of the authors' knowledge, no prior work has investigated automatically building BTs from examples of expert behavior.

Probably the most closely related work to ours involves automatically learning domain-specific planners from example plans [23]. These domain-specific planners are static structures for solving specific planning problems, and are made up of programming components such as loops and conditionals, combined with planning operators. The system is provided with accurate action models in order to build the plans, and implicitly assumes fully observable, deterministic domains.

III. Problem

We propose a problem definition that relaxes the assumptions of the classical planning restricted model (as defined in [29]) in order to more closely reflect the real world. In this problem there are a potentially infinite number of states, which may be partial observations of the complete system. The system may be nondeterministic and may change without agent actions. Actions may occur in parallel, may have a duration, and need not occur at fixed intervals. A policy is learned instead of action models or a plan library, in order to allow robust reactive behaviour in a dynamic environment without expensive replanning [15].

However, we do restrict the problem to learning to carry out a single task or achieve a single goal that is being carried out in the examples, instead of the more general automated planning requirement of being able to form a plan for any specified goal. This reduces the burden on the learner so that it is not forced to depend upon accurate action models for these complex domains. Thus, we define the problem of learning a single task by observation:

Given a set of examples of experts carrying out a single high-level task, $\{E_1, E_2, \ldots, E_n\}$

Where an example is a sequence of cases ordered by time, $E_i = (C_{i1}, C_{i2}, \ldots, C_{im})$

Where a case is an observation and action pair, $C_{ij} = (O_{ij}, A_{ij})$

Where an observation and an action are arbitrary information available to the agent, (eg. a key-value mapping)

Given a similarity metric between pairs of observations and pairs of actions, $M(O_{ij}, O_{kl}) \in [0, 1]$ and $M(A_{ij}, A_{kl}) \in [0, 1]$

Find a policy that will decide the next action given previous cases and the current observations, $\pi((C_{i1}, C_{i2}, \ldots, C_{ij-1}), O_{ij}) \rightarrow A_{ij}$

This policy should be able to reproduce the input action sequences and generalise well to unseen action sequences

This policy should have low run-time cost for selecting actions so that it is applicable for embedded or real-time applications

Note that no information is given about the preconditions or effects of actions, or any conceptual reasoning or task structure behind groups of cases. There is also limited information about failure, as possible actions considered but unused by experts will not be observed, and subsequences of actions which had a negative outcome are observed just like other actions. Experts are assumed to have made appropriate actions, but there may not be one optimal action for a given situation.

IV. The Domain

The domain motivating our problem is the Real-Time Strategy (RTS) video game StarCraft. RTS games are essentially a simplified military simulation, in which players indirectly control many units to gather resources, build infrastructure and armies, and manage units in battle against an opponent. RTS games present some of the toughest challenges for AI agents, making it a difficult area for developing competent AI [30]. It is a particularly attractive area for AI research because of how human players can quickly become adept at dealing with the complexity of the game, with experienced humans outplaying even the best agents from both industry and academia [31].

StarCraft is a very popular RTS game which has recently been increasingly used as a platform for AI research [5]. Due to the popularity of StarCraft, there are many expert players available to provide knowledge and examples of play, and it also has the advantage of the Brood War Application Programming Interface (BWAPI), which provides a way for external code to programmatically query the game state and execute actions as if they were a player in a match. In terms of complexity, StarCraft has real-time constraints, hidden information, minor nondeterminism, long-term goals, multiple levels of abstraction and reasoning, a vast space of actions and game states, durative actions, and long-term action effects [3], [30]–[32]. In order to make the domain slightly more manageable, we have chosen to deal with only the strategic-level actions: build, train, morph, research, and upgrade actions. We also assume that only successfully executed actions are shown, not all inputs from the human, because in the game of StarCraft most professional players very rapidly repeat action inputs until they are executed in order to make actions execute as soon as possible.

V. Behavior Trees

As mentioned earlier in the paper, Behavior Trees (BTs) are being used to represent and act upon the knowledge learned by our system, so this section provides a short overview of BTs. BTs have a hierarchical structure in which top levels generally represent abstract tasks, and subtrees represent different subtasks and behavior for achieving each task. Deeper subtrees represent increasingly-specific behaviors, and leaf nodes represent conditions and primitive actions that interact with the agent's environment (Fig. 1). Although conceptually represented as trees, it is common for task behaviors to be reused at different places in the tree, so the resulting structure is really a directed acyclic graph [6].

Execution of a BT is essentially a depth-first traversal of the directed graph structure, but there are four main non-leaf

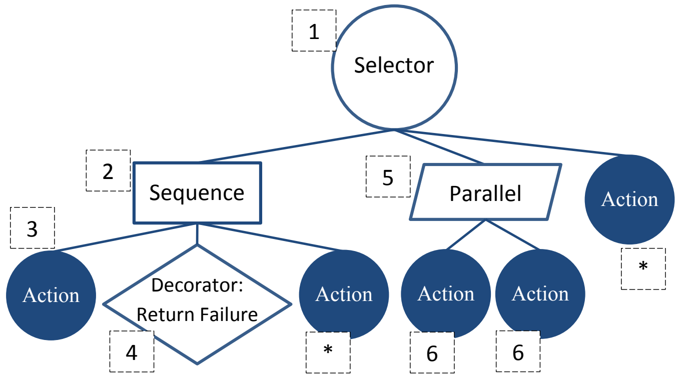

Fig. 1. An example BT, showing the order in which each node would execute. Asterisks indicate nodes which are not executed. Execution begins at the root *selector* node. Next the *sequence* node begins execution – assuming the leftmost child is selected first – and executes its children until a failure is returned by the *decorator* node. The *sequence* node returns a failure and the *selector* node executes its next child. The *parallel* node executes both children simultaneously and successfully returns, allowing the *selector* to return successfully.

node types that control the flow of execution in a BT: sequence, selector, parallel, and decorator nodes [7]. Each node type has a different effect on the execution of its children, and respond differently to failures reported by their children. Sequence nodes run their children in sequence, and usually return with a failure status if any of their children fail. Selector nodes run their children in a priority order, switching to the next child if one of their children fails, and usually return with a success status if any of their children succeed. Selector nodes may alternatively be set to cancel the execution of a child if a higher-priority child becomes executable. Parallel nodes run all their children in parallel, and usually return with a success status if a certain number of their children succeed, or a failure status if a certain number of their children fail. Finally, decorator nodes add extra modifiers or logical conditions to other nodes, for example always returning a success status, or executing only when it has not run before. The specific behavior and even types of nodes can vary depending on the needs of the user.

VI. Method

The first stage in being able to build BTs is to be able to locate areas of commonality within action sequences, as these likely represent common or repeated sub-behaviors. The overall method for creating the behavior tree is an iterative process as follows (Fig. 2). First, a maximally specific BT is created from the given example case sequences. The BT is then iteratively reduced in size by finding and combining common patterns of actions. When no new satisfactory patterns are found, the process stops. By merging similar action patterns, we are forced to generalise the BT and can find where common patterns diverge so we can attempt to infer the reasons for different actions being chosen. Reducing the size of the BT will also help to make it more understandable if people wish to read and edit it.[4]

[4]The code implementation of this method is avilable online at https://github.com/phoglenix/bt-builder.

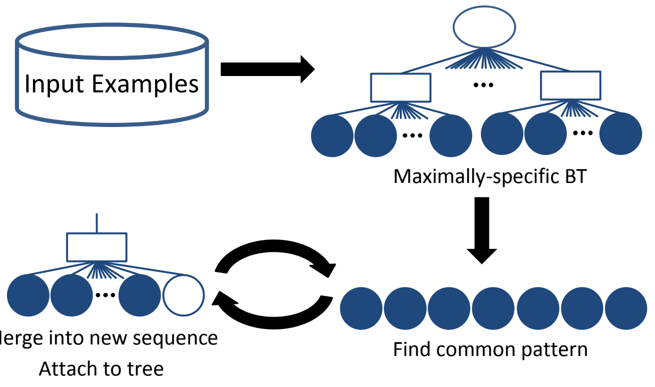

Fig. 2. Overview of the general BT construction process. Input examples are converted into a *maximally-specific BT*. The BT is then iteratively reduced by finding common patterns, merging them into new sequences, and attaching them to the tree. When no more patterns are found, the process stops.

A. Creating the original BT

The process of creating a maximally-specific BT from a set of examples is actually fairly trivial. All actions in an example can simply be made children of a single sequence node (potentially with a special "delay" action between them if timing is known). All of these sequence nodes can then be joined by adding a selector node as their parent to make a complete BT. The selector node can be set to choose randomly among its children, or its children compared with the current state using the observations similarity metric at runtime to select the most-similar option. We call this tree maximally-specific because it exactly represents the input example sequences without any generalisation or other processing. This tree is clearly extremely over-fit to the example data, so it needs to be reduced by finding common patterns.

B. Reducing the BT

Now we iteratively reduce the tree by identifying and merging common subsequences within the sequence nodes and rearranging the tree to share these common sections (Fig. 3). The core of the BT reducing method relies on local sequence alignment techniques. These techniques are commonly used to compare two strings, especially DNA strings in computational biology, to find the indices at which one string aligns best with another. The best alignment is defined by a scoring system that rewards matching or similar characters at a position, penalises mismatching characters, and, importantly, allows but penalises extra or missing characters. For strings of length m and n, efficient implementations of this algorithm run in $O(mn)$ time. In order for this algorithm to be used in our situation, we can extract sequences of actions on different branches of the BT and compare them using the similarity metric given in the problem statement as a scoring system.

While the local alignment algorithm is effective for aligning entire sequences against one another, in this case we are trying to find similar subsequences in cases where the sequence as a whole may not be similar. For this task we can make use of another technique: motif finding. Specifically, we use the Gapped Local Alignment of Motifs (GLAM2) software [33]. This software uses a simulated annealing-based approach to gradually select and refine a short pattern that matches well (scores highly when locally aligned) to many sequences at once. The BT is converted into sequences by simply taking each sequence node separately, and passed to GLAM2. When a pattern has not been improved by GLAM2 for a set number of iterations, it is returned along with the alignments and scores for each sequence. GLAM2 always returns a pattern, so the quality of the pattern and alignments must be checked. We check that all aligned sections have a score above a set threshold, and any alignments with a score below the threshold are discarded. This threshold can be set by informally testing and inspecting the alignments and scores, or can be more rigorously informed by shuffling the input sequences and checking the scores found, or concatenating sequences with shuffled versions of themselves and checking the alignments are more often in the unshuffled regions [33]. If no aligned sequences have a score above the threshold, the BT reduction process stops.

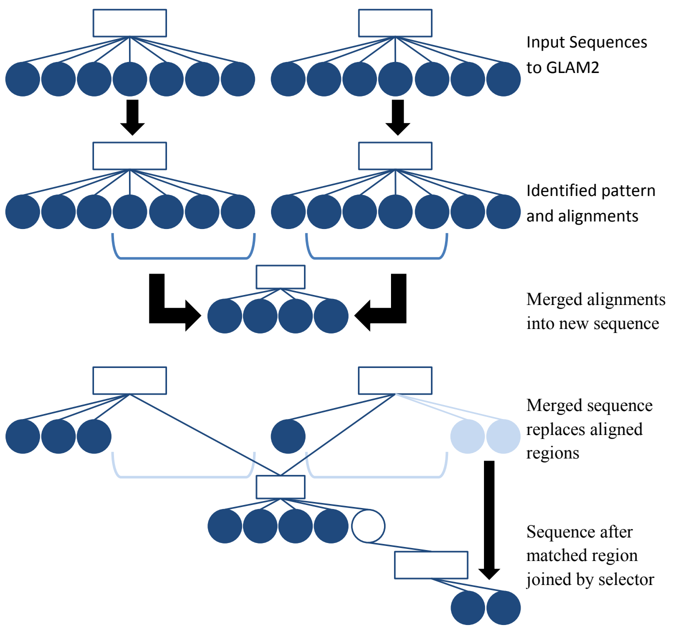
Input Sequences to GLAM2
Identified pattern and alignments
Merged alignments into new sequence
Merged sequence replaces aligned regions
Sequence after matched region joined by selector

Fig. 3. Reducing the BT. Sequences are passed in to GLAM2 for pattern discovery and alignment. Aligned regions are then merged to form a new sequence. The merged sequence then replaces the aligned regions of the original patterns. Finally, any sequences following the aligned region are joined by a selector node.

Using the pattern and alignments found by GLAM2, we begin to construct a new sequence node. This sequence is a generalisation of all of the matching aligned sections of the sequences. For each position in the matched pattern, all nodes at that position in the alignment are merged. This merging produces a weighted combination of the attributes of the nodes. For example, if five nodes had an action with a "name" attribute set to "Train Protoss Probe" and two with "Train Protoss Zealot", the merged node would have a "name" attribute with "Train Protoss Probe"$\times 5$ and "Train Protoss Zealot"$\times 2$. Any attribute values that were seen in just one node of the merge are discarded, because they likely represent unique identifiers or unusual values.

Next, insertion and deletion positions in the sequence are checked for possible transpositions, where actions have occurred in different orders in different sequences. These are detected if an action is almost always inserted either before or after another sequence of actions, but not both (or equivalently, deleted from the pattern and inserted somewhere nearby). In these cases, a parallel node is added with the transposed action (or action sequence) as one child and the sequence it would move around as another child. In cases where insertions and deletions are not detected as parallel or unordered, conditional decorators are added with records of the state observations. When executing these actions, the decorator will be able to check the stored and current state observations in order to decide whether to execute, based on the similarity metric.

Finally, the newly constructed sequence can be used to replace the aligned regions of the original sequences. For each matched sequence, the region before and after the aligned region are separated. For each section before the aligned region, the new sequence is added as the final node. Next, a selector node is added to the end of the newly constructed sequence. For each section after the aligned region, the section is added as a child of the selector node. This will allow the node to select the sequence with the most-similar state observations at execution time. At this point, each sequence node in the tree is passed back to GLAM2 for analysis. Because the previously-found pattern has been collapsed into one sequence node, it will be far less common, so a new pattern will be found.

VII. Experiment

In order to experiment with building behavior trees for StarCraft, we used an existing dataset from prior work [34] consisting of 389 matches of expert human players using the "Protoss" team against another expert "Protoss" team player. The matches were recorded from each player's perspective for a total of 778 examples. The game state observations were sampled every second, and actions were paired with the most-recent observations to produce cases. A very simple action similarity metric was used, producing a score of 1 if the action names were the same (for example "Train Protoss Probe"), and 0 otherwise.

Although there is nothing fundamentally preventing this algorithm being implemented with support for an arbitrary similarity metric between actions or observations, GLAM2 operates only on strings. This meant that the action sequences needed to be encoded as characters in an extended character set before being run in GLAM2, and the results decoded in order to be used as actions again. Even so, GLAM2 was able to find clear motifs in the dataset and the process was able drastically reduce the total number of nodes required to represent the tree, from 218,832 in the original tree, down to 71,294 in the final tree (Fig. 4). The opening actions in each game, in particular, were always discovered as a strong motif early on in the process, as these are very similar every game.

The resulting trees have not yet been tested at actually playing the game, because they would be unable to perform the required low level unit commands that would be required to complement the high-level strategic commands. These responsibilities are separated out into different modules in many StarCraft bots due to the difficulty of multi-scale reasoning [35]. This may be possible to test in future by using the low-level unit control modules from an existing StarCraft bot such as one of [36] or [37].

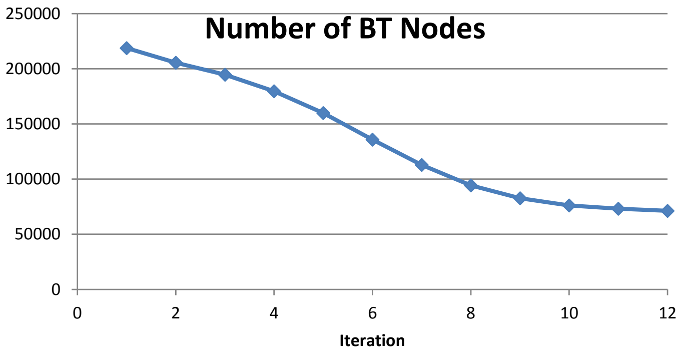

Fig. 4. Number of nodes in the BT throughout a run reducing the StarCraft "Protoss vs Protoss" dataset.

VIII. Discussion and Future Work

This paper presents a promising start to automatically producing a reactive AI system from expert examples, but this approach clearly has some significant limitations. Despite the approach managing to collapse large amounts of repeated or similar sequences of actions, it is not sophisticated enough to separate out most parallel or reactive actions. Parallel actions can be seen in some motifs, in which certain actions are placed before or after other actions in the motif. Currently, GLAM2 cannot detect transpositions, so such variability appears in the alignment as one or more insertion and deletion pairs. This could potentially be mitigated by searching matched regions for inserted and deleted items appearing at a similar frequency across all matched regions, or by analysing the action sequences for ordering relations or a lack thereof. Reactive actions are somewhat captured by the context-sensitive selector nodes that are inserted after each merged region, but the trees produced by this method do not make good use of the behavior tree's potential for reactive behavior. Certain motifs found may very well be reactions to conditions in a game, but they are currently treated as if they are part of the normal sequence of actions, instead of an interruption to the normal sequence. Analysis of the observed conditions leading up to each discovered motif could allow the addition of conditional nodes to trigger the reactive behavior dynamically, which would make the BT much more robust to changes as well as becoming a better and more compact representation. Ideally, we could continuously process the tree to have fewer and fewer nodes while still representing the original information, similar to [19] or [23].

The way in which the trees are reduced necessarily removes information, so it is possible that important information is lost in the process. This is particularly true of the steps in which patterns are used to merge nodes and subsequently join the sequence back to the original locations of the matched regions. In the merging process, only unique attribute values are discarded, but more attention could be paid to generalising these values. Numerical values, in particular, may often be unique, but could be generalised to a range or distribution. There may also be correlations between attributes, which are lost if multiple values for those attributes are merged. This situation might actually be an indication that the node should be merged into multiple nodes instead of just one. For the joining of merged regions back to subsequent behavior, this could potentially break or incorrectly connect longer sequences for which the merged region was just an interruption. This issue would be solved with better identification of reactive actions, as discussed above.

A limitation in the way GLAM2 works is that it always finds at most one pattern match per input sequence. This means that sequences may have to be split up before a pattern that repeats within one sequence will become common enough to be found as the most prominent motif. A related issue is that the algorithm becomes less effective at finding motifs as sequences get shorter and more numerous, which is what happens naturally as they are broken up by the tree-building process. This clearly isn't too major of a problem, because GLAM2 is still able to find motifs quite effectively for many iterations, but it does limit its usefulness. Finally GLAM2 complicates the process due to its use of character encodings for comparison, instead of a more flexible similarity metric. This is understandable, as it was designed to work with DNA and nucleotide sequences and is being extended to work in this scenario.

A useful extension to this work would be to integrate an unsupervised data mining approach to inferring action preconditions and effects, such as [38]. Even a partial understanding of the preconditions and effects of actions could help to guide the BT building process, without having to strongly rely on accurate action models like in HTN planning. As an addition to the problem, or possibly an alternative to the similarity metric, a fitness metric could be provided to the agent to encourage a more search-based strategy of learning. Additionally, a goal description could potentially be supplied in some form to make the problem easier, or a similarity metric could be left for the agent to infer. As an evaluation mechanism, the similarity metric could potentially be used to gauge how close a proposed action was to the expected action, because it may be impossible to predict the exact action details used by the expert.

IX. Conclusion

In this paper we introduced a new planning problem, in which no action models were supplied and very few assumptions were made about the domain. In this problem, the planning system is able to receive information about the domain only through observing examples of expert behavior in the domain, and must be able to complete the same task that the experts were undertaking in the examples. We introduced behavior trees as a potential mechanism for representing and executing knowledge about the problem and appropriate actions to take to complete the task. We then introduced our mechanism for producing a solution behavior tree, which involves searching for common motifs among sequences of actions, joining the sequences found, and following them with selector nodes that allow some reactivity to the current game state. The behavior tree learning mechanism was shown to successfully reduce sequences of player actions from the real-time strategy game StarCraft by almost two orders of magnitude.

References

[1] O. Ilghami, D. S. Nau, H. Muñoz-Avila, and D. W. Aha, "Learning preconditions for planning from plan traces and htn structure," *Computational Intelligence*, vol. 21, no. 4, pp. 388–413, 2005.

[2] J. Lanchas, S. Jiménez, F. Fernández, and D. Borrajo, "Learning action durations from executions," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS) Confer*, 2007.

[3] S. Ontañón, "Case acquisition strategies for case-based reasoning in real-time strategy games," in *Proceedings of the International Florida Artificial Intelligence Research Society (FLAIRS) Conference*, 2012.

[4] G. Florez-Puga, M. Gomez-Martin, P. Gomez-Martin, B. Diaz-Agudo, and P. Gonzalez-Calero, "Query-enabled behavior trees," *IEEE Trans. Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 298–308, Dec 2009.

[5] G. Robertson and I. Watson, "A review of real-time strategy game AI," *AI Magazine*, vol. 35, no. 4, pp. 75–104, 2014.

[6] D. Isla, "Proceedings of the game developers conference: Handling complexity in the Halo 2 AI," Web page, March 2005. [Online]. Available: http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

[7] A. Champandard, "Getting started with decision making and control systems," in *AI Game Programming Wisdom*. Charles River Media, 2008, vol. 4, pp. 257–264.

[8] R. Palma, P. González-Calero, M. Gómez-Martín, and P. Gómez-Martín, "Extending case-based planning with behavior trees," in *Proceedings of the International FLAIRS Conference*, 2011, pp. 407–412.

[9] A. J. Champandard, "Behavior trees for next-gen game AI," Video, December 2007, retrieved 15 November 2012. [Online]. Available: http://aigamedev.com/open/article/behavior-trees-part1/

[10] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3-4, pp. 189 – 208, 1971.

[11] X. Wang, "Learning by observation and practice: An incremental approach for planning operator acquisition," in *Proceedings of the International Conference on Machine Learning (ICML)*, 1995, pp. 549–557.

[12] Q. Yang, K. Wu, and Y. Jiang, "Learning action models from plan examples using weighted max-sat," *Artificial Intelligence*, vol. 171, no. 2, pp. 107–143, 2007.

[13] H. H. Zhuo, D. H. Hu, C. Hogg, Q. Yang, and H. Munoz-Avila, "Learning htn method preconditions and action models from partial observations," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2009, pp. 1804–1810.

[14] C. Hogg, H. Munoz-Avila, and U. Kuter, "HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required," in *Proceedings of the AAAI Conference on AI*, 2008, pp. 950–956.

[15] S. Mohan and J. E. Laird, "Learning goal-oriented hierarchical tasks from situated interactive instruction," in *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI) Conference*, 2014.

[16] N. Mehta, "Hierarchical structure discovery and transfer in sequential decision problems," Ph.D. dissertation, Oregon State University, 2011.

[17] N. Nejati, P. Langley, and T. Konik, "Learning hierarchical task networks by observation," in *Proceedings of the International Conference on Machine Learning*, 2006, pp. 665–672.

[18] C. Hogg, U. Kuter, and H. Munoz-Avila, "Learning hierarchical task networks for nondeterministic planning domains," in *Proceedings of the IJCAI*, 2009.

[19] H. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning probabilistic relational planning rules." in *Proceedings of the ICAPS Conference*, 2004, pp. 73–82.

[20] M. D. Schmill, T. Oates, and P. R. Cohen, "Learning planning operators in real-world, partially observable environments." in *Proceedings of the Artificial Intelligence Planning and Scheduling Conference*, 2000, pp. 246–253.

[21] D. Shahaf and E. Amir, "Learning partially observable action schemas," in *Proceedings of the AAAI Conference*, 2006, pp. 913–919.

[22] Q. Yang, K. Wu, and Y. Jiang, “Learning actions models from plan examples with incomplete knowledge,” in *Proceedings of the ICAPS Conference*, 2005, pp. 241–250.

[23] E. Winner and M. Veloso, “Distill: Learning domain-specific planners by example,” in *Proceedings of the International Conference on Machine Learning*, 2003, pp. 800–807.

[24] R. Palma, A. Sánchez-Ruiz, M. Gómez-Martín, P. Gómez-Martín, and P. González-Calero, “Combining expert knowledge and learning from demonstration in real-time strategy games,” in *Case-Based Reasoning Research and Development*, ser. Lecture Notes in Computer Science, A. Ram and N. Wiratunga, Eds. Springer Berlin / Heidelberg, 2011, vol. 6880, pp. 181–195.

[25] E. Dereszynski, J. Hostetler, A. Fern, T. Dietterich, T. Hoang, and M. Udarbe, “Learning probabilistic behavior models in real-time strategy games,” in *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment (AIIDE) Conference*. AAAI Press, 2011, pp. 20–25.

[26] G. Synnaeve and P. Bessière, “A bayesian model for plan recognition in RTS games applied to StarCraft,” in *Proceedings of the AIIDE Conference*. AAAI Press, 2011, pp. 79–84.

[27] C. Lim, R. Baumgarten, and S. Colton, “Evolving behaviour trees for the commercial game DEFCON,” in *Applications of Evolutionary Computation*, ser. Lecture Notes in Computer Science, C. Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcazar, C.-K. Goh, J. Merelo, F. Neri, M. Preuß, J. Togelius, and G. Yannakakis, Eds. Springer Berlin / Heidelberg, 2010, vol. 6024, pp. 100–110.

[28] R. Kadlec, “Evolution of intelligent agent behavior in computer games,” Master’s thesis, Faculty of Mathematics and Physics, Charles University in Prague, 2008.

[29] M. Ghallab, D. Nau, and P. Traverso, *Automated planning: theory & practice*. Elsevier, 2004, chapter 11.

[30] M. Buro and T. M. Furtak, “RTS games and real-time AI research,” in *Proceedings of the Behavior Representation in Modeling and Simulation Conference*. Citeseer, 2004, pp. 63–70.

[31] M. Buro and D. Churchill, “Real-time strategy game competitions,” *AI Magazine*, vol. 33, no. 3, pp. 106–108, Fall 2012.

[32] B. Weber, M. Mateas, and A. Jhala, “Building human-level AI for real-time strategy games,” in *Proceedings of the AAAI Fall Symposium Series*. AAAI, 2011, pp. 329–336.

[33] M. C. Frith, N. F. W. Saunders, B. Kobe, and T. L. Bailey, “Discovering sequence motifs with arbitrary insertions and deletions,” *PLoS Computational Biology*, vol. 4, no. 5, p. e1000071, 2008.

[34] G. Robertson and I. Watson, “An improved dataset and extraction process for StarCraft AI,” in *Proceedings of the FLAIRS Conference*, 2014.

[35] B. Weber, P. Mawhorter, M. Mateas, and A. Jhala, “Reactive planning idioms for multi-scale game AI,” in *Proceedings of the IEEE Conference on Computational Intelligence and Games*. IEEE, 2010, pp. 115–122.

[36] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, “A survey of real-time strategy game AI research and competition in StarCraft,” *IEEE Trans. Computational Intelligence and AI in Games*, vol. 5, no. 4, pp. 293–311, 2013.

[37] S. Wender and I. Watson, “Integrating case-based reasoning with reinforcement learning for real-time strategy game micromanagement,” in *PRICAI 2014: Trends in Artificial Intelligence*. Springer, 2014, pp. 64–76.

[38] M. A. Leece and A. Jhala, “Sequential pattern mining in StarCraft: Brood War for short and long-term goals,” in *In Proceedings of the AIIDE Conference*, 2014.