Meta-Level Behavior Adaptation in Real-Time Strategy Games

Manish Mehta[1], Santiago Ontañón[2], and Ashwin Ram[1]

[1] CCL, Cognitive Computing Lab Georgia Institute of Technology,
Atlanta, GA 303322/0280,
{mehtama1,ashwin}@cc.gatech.edu
[2] IIIA, Artificial Intelligence Research Institute
CSIC, Spanish Council for Scientific Research
Campus UAB, 08193 Bellaterra (Spain),
santi@iiia.csic.es

Abstract. AI agents designed for real-time settings need to adapt themselves to changing circumstances to improve their performance and remedy their faults. Agents typically designed for computer games, however, lack this ability. The lack of adaptivity causes a break in player experience when they repeatedly fail to behave properly in circumstances unforeseen by the game designers. In this paper, we focus on an AI technique for game playing agents that helps them adapt to changing game circumstances. The agents carry out runtime adaptation of their behavior sets by monitoring and reasoning about their behavior execution and use this reasoning to dynamically carry out revisions on the behaviors. The evaluation of the behavior adaptation approach in a complex real-time strategy game shows that the agents adapt themselves and improve their performance by revising their behavior sets appropriately.

1 Introduction

Faults during a problem solving episode provide both humans and artificial agents strong cues on what needs to be learned [4]. AI agents can thus benefit from approaches that provide them the ability to learn from their failed encounters in the world. The need for adaptivity for agents that play modern computer games has been emphasized as well [11]. This paper presents an approach that addresses this need for AI game playing agents to be adaptive, learning from their experiences to improve their performance.

AI agents for games are typically created using hand authored behaviors that are static in nature [10]. This results in agents that cannot dynamically adapt themselves to changing scenarios within the game. Incorporating knowledge representation and planning techniques can enable an agent's behavior to become more flexible in novel situations, but it does not guarantee success: when an agent's behavior fails to achieve its desired purpose, most agents are unable to identify such failure and will continue executing the ineffective behavior. Ideally we need self-adapting AI agents for games that can learn from their own experience and adapt themselves. The problem is harder for state of the art real

time games. Most of them involve complex strategic behaviors, such as real-time strategy (RTS) games, or involve characters that must behave human-like in a believable way. Both kind of games have huge decision spaces and require real time performance which complicate the creation of adaptive AI approaches.

In this paper, we address the problem of lack of adaptivity in RTS game playing agents. Our proposal involves the use of meta-reasoning [3] to identify the execution failures and their causes and carrying out appropriate revisions in response. We have developed the meta-reasoning approach to address two classes of failures. The first class of failures are based on deliberating over the analyzed differences across successful and failed executions and appropriately addressing the important differences. The second class of failures represent anomalous situations that can be identified from a game playing episode and need to be appropriately addressed. In our previous work [7], we focused explicitly on addressing the first class of failures and fixes needed to resolve the analyzed differences (across successful and failed executions) and the corresponding experimental evaluation looked at the performance of a specific game playing system, Darmok [13], with and without the meta-reasoning layer.

In the work presented in this paper, our focus is on the second class of failures. Our approach is based on using a collection of author-defined *failure patterns*, which are used to represent explicit descriptions of anomalous situations. Failure patterns provide a case-based solution for detecting failures in the execution of the base reasoner. After failures have been detected, fixes are made in the system to prevent those failures from happening again. These failure patterns explain what went wrong with the executed actions in the world (and not just differences across successful and failed system executions). This paper has three main contributions. First, we present a meta-reasoning approach that can be used in real-time game domains with limited number of trials to improve the performance. Second, instead of directly modifying the behaviors being run by the game playing agent, our meta-reasoning module creates *daemons*, which operate over the executing behaviors in real time to detect and prevent the anomalous situations from happening again. Third, and most important, we propose to use *failure patterns*. Instead of designing behavior sets which can generate correct behavior in all situations, failure patterns allow the game designer to specify a set of anomalous situations, and associate possible fixes for them (failure patterns). We argue that this approach alleviates the behavior creation process, and, coupled with the proposed meta-reasoning module, achieves adaptive behavior for real-time games.

Our test results show that our proposal improves the performance over different game playing sessions. Meta-reasoning systems are composed of a base reasoner that is in charge of the performance task (in this case playing an RTS game), and a meta-reasoner that observes and modifies the base reasoner's behavior. As in our previous work [7], in this paper we are going to use Darmok [9] as the base reasoning system, that is a case-based planning system designed to play RTS games. The resulting system by applying our meta-reasoning approach to Darmok is called Meta-Darmok.

The rest of the paper is organized as follows. First we present the related work. Then, we introduce Darmok and the RTS game used in our experiments. After that, we present our meta-adaptation approach, the Meta-Darmok system, and the empirical evaluation of it. Finally we conclude the paper and discuss future steps.

2 Related Work

Meta-reasoning [3] is the process of monitoring and control of reasoning. The control part involves the meta-level control of the computational resources spend in specific tasks (attention). The monitoring part involves figuring out what went wrong and why through *introspective reasoning*. Our system explores the later aspect of meta-reasoning. The Meta-Aqua system, for example, uses a library of pre-defined patterns of erroneous interactions among reasoning steps to recognize failures in the story understanding task [4]. The Autognostic system [12] uses a model of the system to localize the error in the system's element and and uses the model to construct a possible alternative trace which could lead to the desired but unaccomplished solution. IULIAN [8] uses questions about its own reasoning and knowledge to re-index its memory and to regulate its processing. These approaches typically assume the agent is the sole source of change, actions are deterministic, take unit time, and their effects well defined, and that the world is fully observable. In RTS games all these assumptions are violated. Furthermore, these approaches are generally applied to detect failures in plans consisting of STRIPS operators or minor extensions of STRIPS; using it for revising complex behavior sets for game domains requires novel failure detection techniques, a vocabulary of failure patterns pertinent to game domains and new behavior modification strategies.

One of the approaches to planning that deal best with exogenous events and non-determinism are those based on Markov decision processes (MDP), focusing on learning a policy. These approaches, however, require a large number of iterations to converge and only do so if certain conditions are met. In complex game domains, these techniques are intractable (MDP learning algorithms require a polynomial time in the number of states of a problem [5], which in the case of complex games is prohibitively large). Further, these approaches generalize poorly. The playing style of a human player can significantly change the game dynamics; the learned static policy might have to be retrained to accommodate such changes.

More recent work has relaxed these assumptions and has been applied to more complex game domains [2, 14]. Another body of related work applied to complex game domains is in Adaptive AI [11]. Dynamic Scripting is a technique based on reinforcement learning, that is able to generate "scripts" by drawing subsets of rules from a pre-authored large collection of rules. If the rules are properly authored, different subsets of those rules provide meaningful and different behaviors. Dynamic scripting is a technique for learning which subsets of

those rules work better under different circumstances, by learning which rules work good or bad, altering their probability of being selected again.

3 Application Domain: WARGUS and Darmok

We have used WARGUS, a real time strategy game, as our game domain. Each player's goal in WARGUS is to survive and destroy the other players. Each player has a number of troops, buildings, and workers (who gather resources such as gold, wood and oil in order to produce more units). Buildings are required to produce more advanced troops, and troops are required to attack the enemy. The calculations inherent in the combat system make the game non-deterministic. The game involves complex strategic reasoning, such as terrain analysis, resource handling, planning, and scheduling, all of them under tight time constraints. WARGUS (sometimes referred to as "stratagus") is a well known domain, and has been used by several researchers as a test bed for AI techniques [1, 6].

Darmok is a real time planning and execution system that has been designed to play games such as WARGUS and is capable of dealing with both the vast decision spaces and the real-time component of RTS games. [9]. Darmok combines learning from demonstration with case-based planning:

- *Learning from Demonstration*: Darmok learns behaviors (stored as cases to be later used using case-based planning), by analyzing annotated game traces. An annotated game trace is a log of the actions executed by a human expert to play and win a game, which includes annotations of which goals were pursued with each action.
- *Case-based Planning*: Darmok uses the behaviors learnt from demonstration through case-based planning. Cases are retrieved, executed and adapted on real time. When plans fail, they are retracted, and new adaptations or new cases are attempted.

Darmok plays the complete game without any abstraction, and it takes every single decision in the game, beginning to end. Moreover, Darmok's degree of proficiency depends highly on the demonstrations being provided. If good demonstrations are provided, Darmok can learn to defeat the built-in AI of WARGUS in a variety of maps. Darmok is capable of taking behaviors observed from a human in a demonstration and adapt them for different situations. However, due to the complexity of the domain, this adaptation is a complex procedure, and Darmok does not always succeed in producing a good enough adaptation. Moreover, Darmok has to decide which behaviors, from the set of behaviors observed from humans, to use in each map. This selection is performed by both assessing game state similarity and goal similarity, however, sometimes Darmok fails to assess the current situation, and a suboptimal behavior is selected. The meta-level behavior adaptation approach presented in this paper uses Darmok as the base reasoning system to execute the behaviors that play the game, and monitors the behavior of Darmok, in order to fix it when failures are detected.

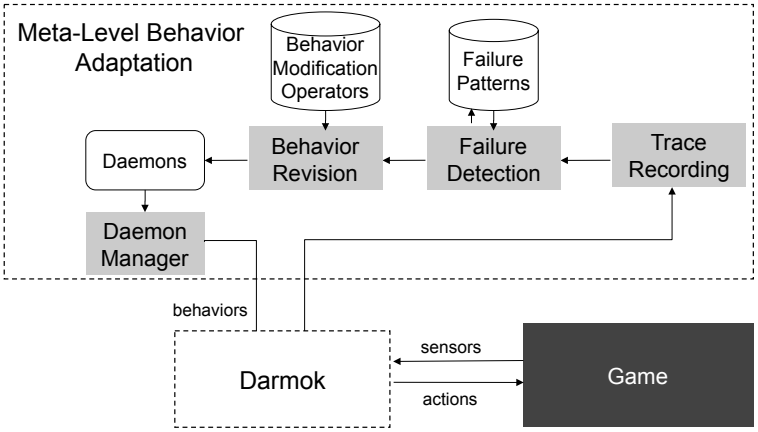

Fig. 1. The proposed behavior modification architecture.

4 Meta-Level Behavior Adaptation

Once it has learnt, the Darmok system is not able to modify the behaviors it has learnt. Although Darmok has the capability of learning from experience (by remembering which behaviors succeeded and which ones failed in different situations), it does not have any capability to fix the behaviors in its case base. If the expert that Darmok learnt from made a mistake in one of the behaviors, Darmok will repeat that mistake again and again each time Darmok retrieves that behavior. The meta-reasoning approach presented in this paper provides Darmok exactly with that capability, resulting in a system called *Meta-Darmok*, shown in Figure 1. By analyzing past performances, Meta-Darmok can fix the behavior resulting from the behaviors in its case base.

Our meta-level behavior adaptation approach consists of four parts: *Trace Recording*, *Failure Detection*, *Behavior Modification*, and the *Daemon Manager*. During trace recording, a trace holding important events happening during the game is recorded. Failure detection involves analyzing the execution trace to find possible issues with the executing behaviors by using a set of *failure patterns*. These failure patterns represent a set of pre-compiled patterns that can identify the cause of each particular failure by identifying instances of these patterns in the trace. Once a set of failures has been identified, the failed conditions can be resolved by appropriately revising the behavior using a set of *behavior modification routines*. These behavior modification routines are created using a combination of basic modification operators (called *modops*, as explained later). The modifications are inserted as *daemons*, which monitor for failure conditions to happen during execution when Darmok retrieves some particular behaviors. A daemon manager triggers the execution of such daemons when required.

4.1 Trace Recording

The behavior execution system during execution records a trace that contains information related to basic events including the name of the behavior that was being executed, the corresponding game state when the event occurred, the time at which the behavior started, failed or succeeded, and the delay from the moment the behavior became ready for execution to the time when it actually started executing. All this information is recorded in the execution trace, which the system updates as events occur at runtime. The trace provides a considerable advantage in performing behavior adaptation with respect to only analyzing the instant in which the failure occurred, since the trace can help localize portions that could possibly have been responsible for the failure. Once a game finishes, an *abstracted trace* is created from the execution trace that Darmok generates. The abstracted trace is the one used by the rest of components of the meta-reasoning module. The abstracted trace, consists of various relevant domain-dependent features: like information regarding units such as hit points, or location, information related to units that were idle, killed or attacked and the cycles at which this happens. The abstracted trace also contains basic behavior failure data that consists of the apparent reason for behavior failures, such as whether it was due to insufficient resources or not having a particular unit available to carry out a behavior. The abstracted trace is used to find occurrences of the failure patterns.

4.2 Failure Detection

Failure detection involves localizing the fault points. Although the abstracted execution trace defines the space of possible causes for the failure, it does not provide any help in localizing the cause of the failure. Traces can be extremely large, especially in the case of complex RTS games on which the system may spend a lot of effort attempting to achieve a particular goal. In order to avoid this potentially very expensive search, a set of pre-compiled patterns of failures can be used to help identify the cause of each particular failure by identifying instances of these patterns in the trace [4]. The failure patterns essentially provide an abstraction mechanism to look for typical patterns of failure conditions over the execution trace. The failure patterns simplify the blame-assignment process into a search for instances of the particular problematic patterns.

Failure patterns are defined as finite state machines (FSMs) that look for generic patterns in the abstracted trace. An example of a failure pattern represented as FSM is *Very Close Resource Gathering Location failure* (VCRGLfail) (shown in Figure 2) that detects whether a peasant is gathering resources at a location that is too close to the enemy compared to other possible resource gathering locations. This could lead to an opening for enemy units to attack early. Other examples of failure patterns and their corresponding behavior modification operators are given in Table 1. Each failure pattern is associated with modification routines. When a failure pattern generates a match in the abstracted trace, an instantiation of the failure pattern is created. Each instantiation contains which were the particular events in the abstracted trace that matched with the

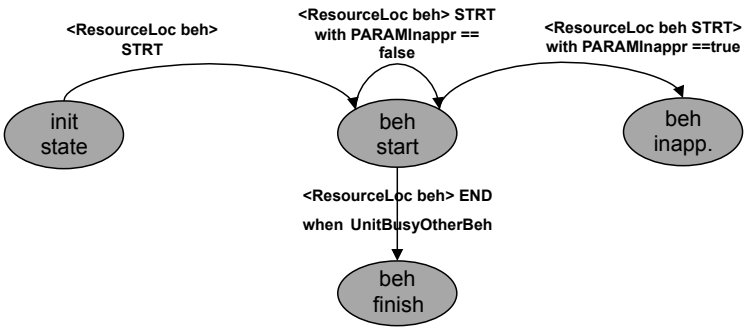

Fig. 2. The figure shows the FSM corresponding to the failure pattern VCRGLfail. STRT, END represent the status of the behavior. PARAMInappr. represents the routine that checks whether the location where peasant is gathering resources is at a location that is too close to the enemy compared to other resource gathering locations.

pattern. This is used to instantiate particular behavior modification routines that are targeted to the particular behaviors that were to blame for the failure.

4.3 Behavior Modification

Once the cause of the failure is identified, it needs to be addressed through appropriate modification. These modifications are in the form of inserting or removing steps at the correct position in the failed behavior, or changing some parameter of an executing behavior. Once the failure patterns are detected from the execution trace, the corresponding behavior modification routines and the failed conditions are inserted as daemons for the map in which these failed conditions are detected. The daemons act as a meta-level reactive plan that operates over the executing behavior at runtime. The conditions for the failure pattern become the preconditions of the daemon and the behavior modification routine consisting of basic modops become the steps to execute when the daemon executes. The daemons operate over the executing behavior, monitor their execution, detect whether a failure is about to happen and repair the behavior according to the defined plan modification routines. Notice that Meta-Darmok does not directly modify the behaviors in the case-base of Darmok, but reactively modifies those behaviors when Darmok is executing them.

In the current system, we have defined 20 failure patterns and behavior modification routines for WARGUS. The way Meta-Darmok improves over time is by accumulating the daemons that the meta-reasoner generates (which are associated to particular maps). Thus, over time, Meta-Darmok improves performance by learning which combination of daemons improves the performance of Darmok for each map. The adaptation system can be easily extended by writing other patterns of failure (as described in [15]) that could be detected from the abstracted trace and the appropriate behavior modifications to the corresponding behaviors that need to be carried out in order to correct the failed situation.

Failure Pattern	Behavior Modification Operator
Resource Idle failure (e.g., resource like peasant, building, enemy units could be idle)	Utilize the resource in a more productive manner (for example, send peasant to gather more resources or use the peasant to create a building that needed later on)
Very Close Resource Gathering Location Failure	Change the location for resource gathering to a more appropriate one
Inappropriate Enemy Attacked failure	Direct the attack towards the more dangerous enemy unit
Inappropriate Attack Location failure	Change the attack location to a more appropriate one

Table 1. Some example failure patterns and their associated behavior modification operators in WARGUS

5 Empirical Evaluation

To evaluate our behavior adaptation approach, we conducted two different experiments turning the behavior adaptation on and off respectively. The experiments were conducted on 8 different variations of the 2-player version of the classical map "Nowhere to run, Nowhere to hide"(NWTR), one of the maps from *Battlenet* regularly player by human players, characterized by a wall of trees that separates the players. This map leads to complex strategic reasoning, such as building long range units (such as catapults or ballistas) to attack the other player before the wall of trees has been destroyed, tunneling early in the game through the wall of trees trying to catch the enemy by surprise, or other strategies. The focus of the experiments is the ability of meta-reasoning layer to detect anomalous situations and revise the behavior sets. The results are reported for 24 games in 6 different scenarios. These 6 scenarios correspond to 6 different expert demonstrations from NWTR maps. As Darmok can learn from more than one demonstration, we evaluate results when Darmok learns from one upto six demonstrations. Each one of the expert demonstrations exemplified different techniques with which the game can be played: fighters rush, knights rush, ranged attacks using ballistas, or blocking the enemy using towers.

Figure 3 shows the average results for all the 144 games in terms of number of wins, draw and losses. The figure also shows average player and opponent score (where the "score" is a number that WARGUS itself calculates and assigns to each player at the end of each game). Finally, WP shows the win percentage, presenting the improvement in win percentage comparing adaptation with respect to no adaptation. The results show that behavior adaptation leads to an improvement of the percentage of wins as well as the player score to opponent score ratio. Figure 4 shows the overall system improvement plotted against the number of traces. An improvement occurs in all cases irrespective of the number of traces used. Overall system performance improved considerably (overall increase of 62.3% on average).

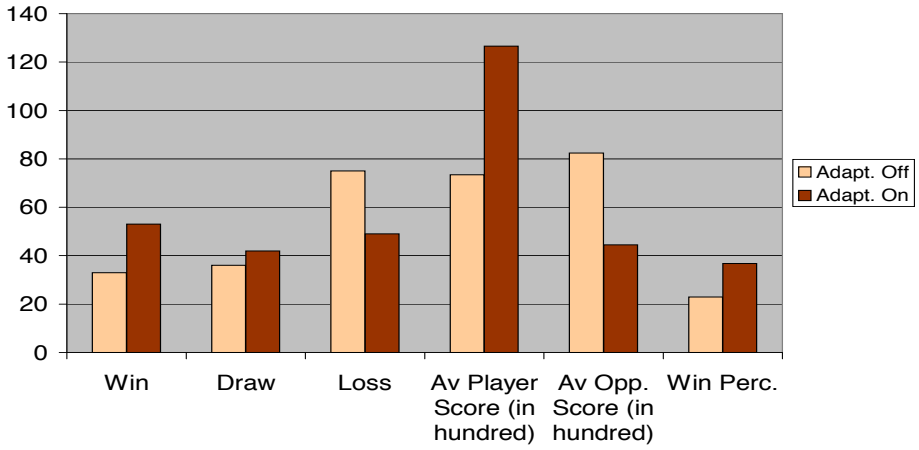

Fig. 3. The figure shows the average results from experiments turning the behavior adaptation on and off

6 Conclusion and Future Work

We aim to create adaptive agents for complex real-time games that learn from their experience and avoid their mistakes. In this paper, we have presented an approach based on meta-reasoning to behavior adaptation that achieves this. It is based on the use of failure patterns in order to ease blame assignment, a vocabulary of failure patterns to detect anomalous situations pertinent to game domains and behavior modification strategies to figure out what went wrong and the reasoning behind it. We have shown that *daemons* can be used to introduce reactive elements in the execution of the system that will adapt the behavior if failures are detected in real time. Our approach has been implemented and tested in a real time strategy game WARGUS. Our experimentation results indicate that overall performance of the system improves with the introduced behavior adaptations. One of the issues in the current behavior modification system is introduction of conflicting changes which can possibly result in degradation of performance. Some of the introduced changes cause unwanted revisions of the behavior causing a degradation in system performance. We plan to address this issue in the future. We intend to run more experiments on different maps to further test our approach and identify other potential issues in the future.

References

1. David Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *ICCBR'2005*, number 3620 in Lecture Notes in Artificial Intelligence, pages 5–20, 2005.
2. Tristan Cazenave. Metarules to improve tactical go knowledge. *Inf. Sci. Inf. Comput. Sci.*, 154(3-4):173–188, 2003.

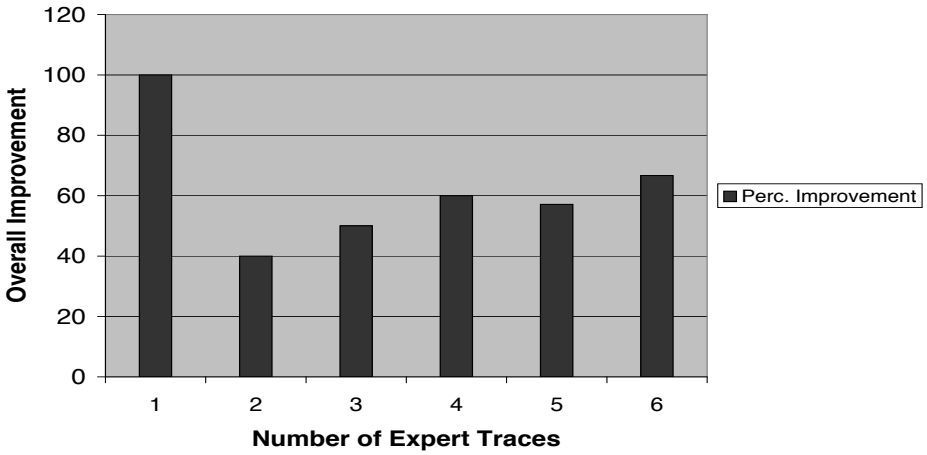

Fig. 4. The figure shows the overall percentage improvement using the behavior adaptation approach

3. Michael T. Cox. Metacognition in computation: a selected research review. *Artif. Intell.*, 169(2):104–141, 2005.
4. Michael T. Cox and Ashwin Ram. Introspective multistrategy learning: On the construction of learning strategies. Technical report, 1996.
5. Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232, 2002.
6. Josh McCoy and Michael Mateas. An integrated agent for playing real-time strategy games. In *AAAI*, pages 1313–1318. AAAI Press, 2008.
7. Manish Mehta, Santiago Ontañón, and Ashwin Ram. Using meta-reasoning to improve the performance of case-based planning. In *ICCBR 2009*, 2009.
8. Rudiger Oehlmann. Metacognitive adaptation: Regulating the plan transformation process. In *Proceedings of the Fall Symposium on Adaptation of Knowledge for Reuse*, 1995.
9. Santiago Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. On-line case-based planning. *Computational Intelligence Journal*, 26(1):84–119, 2010.
10. S. Rabin. *AI Game Programming Wisdom*. Charles River Media, 2002.
11. Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen Kuyper, and Eric Postma. Adaptive game ai with dynamic scripting. *Machine Learning*, 63(3):217–248, 2006.
12. Eleni Stroulia and Ashok K. Goel. Functional representation and reasoning in reflective systems. *Journal of Applied Intelligence*, 9:101–124, 1995.
13. Neha Sugandh, Santiago Ontañón, and Ashwin Ram. Real-time plan adaptation for case-based planning in real-time strategy games. In *ECCBR 2008*, pages 533–547, Berlin, Heidelberg, 2008. Springer-Verlag.
14. Patrick Ulam, Joshua Jones, and Ashok K. Goel. Combining model-based meta-reasoning and reinforcement learning for adapting game-playing agents. In *AIIDE*, 2008.
15. Suhas Virmani, Yatin Kanetkar, Manish Mehta, Santiago Ontañón, and Ashwin Ram. An intelligent IDE for behavior authoring in real-time strategy games. In *AIIDE*, 2008.