Using Simple Games to Evaluate Self-Organization Concepts: a Whack-a-mole Case Study

Nick Nygren
Department of Computer Science
University of Calgary
Calgary, Canada
ndnygren@ucalgary.ca

Jörg Denzinger
Department of Computer Science
University of Calgary
Calgary, Canada
denzinge@cpsc.ucalgary.ca

***Abstract*—We present the idea of using variants of simple games as an easy additional application area to establish generality of AI concepts. We substantiate this idea by using multi-hammer Whack-a-mole as application area for the efficiency improvement advisor and extended efficiency improvement advisor concepts for self-organizing multi-agent systems. Both concepts have previously been applied to pickup-and-delivery problems and were claimed to be general concepts for improving solving dynamic task fulfillment problems. Our experiments with multi-hammer Whack-a-mole show similar improvements to the other area for both concepts, giving credit to the generality claim.**

***Index Terms*—efficiency improvement advisor, extended efficiency improvement advisor, self-adaptation, Whack-a-mole, Multi-agent systems**

I. Introduction

Complex games intended for human players have a long history of being targeted by AI researchers. Some examples are Deep Blue for chess [3], or AlphaGo for Go [8]. Naturally, in addition to beating human players, a main goal in developing game playing AIs is to develop methods that then can also be applied in other applications. In this paper, we have the opposite intend. We look at an existing concept for an industrial application, pickup-and-delivery, and apply it to a slight modification of a well-known and simple human game, multi-hammer Whack-a-mole, in order to establish the *generality* of the concept.

More precisely, we are looking at two variants of a concept for improving the cooperation of several agents that perform dynamic task fulfillment, i.e. they have to perform tasks that might be known ahead of time, can be anticipated or appear "out of the blue". The concept is called the efficiency improvement advisor (EIA, its original variant described in [9], an improvement, the extended efficiency improvement advisor, EEIA, recently published in [7]) and it aims at improving the behavior of self-organizing multi-agent systems by introducing an additional agent, the EIA (or EEIA), that uses various knowledge to adapt the whole multi-agent system to the tasks it has to fulfill over longer periods of time. The EIA uses knowledge about recurring tasks it creates by analyzing the history of the system, while the EEIA adds to this knowledge about anticipated tasks sure knowledge about the appearance of tasks. Both advisor agents use the knowledge to create good solutions for the known and anticipated tasks and then create exception rules for the task fulfilling agents to improve their otherwise autonomous behavior. For pickup-and-delivery problems, both advisors showed substantial improvements when their application conditions were fulfilled.

Multi-hammer Whack-a-mole expands the well-known game Whack-a-mole by allowing for the simultaneous use of several hammers. We also created game settings in which the appearance of some moles is known ahead of time and where some mole appearances persist over a number of games. But the key property that had us select this game is the dynamic nature of the tasks, which is the challenge for the hammers.

Our experiments with the advisors in Whack-a-mole showed similar results as for pickup-and-delivery. The EIA improves the behavior of the hammers (without them having to communicate with each other during a game) and if additional sure knowledge was available the performance with the EEIA was even better. Compared to the pickup-and-delivery domain, creating the Whack-a-mole system (i.e. a game simulator and the behavior of hammers and advisors) was easily done, which justifies our suggestion to use variants of simple games as additional application areas for concepts to establish their generality.

In the following, we first give a short introduction of the EIA and its improvement EEIA in Section II. Then we introduce multi-hammer Whack-a-mole and how it fits into the general problem of dynamic task fulfillment in Section III. In Section IV, we instantiate the advisor concepts to whack-a-mole and then, in Section V, we report on our evaluations of the variants of the advisor instantiated to Whack-a-mole. This if followed, in Section VI, by a discussion of these results and the results from [7] with regard to the generality of the advisor concepts. Section VII provides a short presentation of related work and Section VIII concludes the paper with some final remarks.

978-1-7281-1884-0/19/$31.00

II. EIA AND EEIA

In this section, we briefly describe dynamic task fulfillment (DTF), which represents part of the generality claim for the advisor concept. This is followed by describing said advisor concept (EIA) that uses an analysis of a multi-agent system's past to determine recurring tasks. Badly solved recurring tasks are corrected with rule-based advice to the individual agents to improve the system behavior. We also describe extending the EIA to the EEIA that additionally uses knowledge about known-ahead, pre-arranged tasks for an additional improvement.

A. Dynamic task fulfillment

The problem of dynamic task fulfillment involves a set T of possible tasks that have to be fulfilled in an environment $\mathcal{E}nv$ and a set A of agents that can perform tasks. An instance of a DTF problem is called a *run instance* and has the form

$$((ta_1, t_1), (ta_2, t_2), ..., (ta_m, t_m))$$

with $ta_i \in T$, $t_i \in Time$ (the time interval the task fulfillment has to happen in) and $t_i \leq t_{i+1}$. Usually, in order to allow for an analysis of a system's history to create knowledge, we will look at a sequence of such run instances.

When the agents in A work on a run instance they generate a so-called emergent solution sol, which has the form

$$sol = ((ta'_1, \mathcal{A}g'_1, t'_1), (ta'_2, \mathcal{A}g'_2, t'_2), ..., (ta'_m, \mathcal{A}g'_m, t'_m)),$$

where $ta'_m \in \{ta_1, ..., ta_m\}$, $ta'_i \neq ta'_j$ for all $i \neq j$, $\mathcal{A}g'_i \in A$, $t'_i \leq t'_{i+1}$, $t'_i \in Time$. Here, $(ta'_1, \mathcal{A}g'_1, t'_1)$ means task ta'_i was started by $\mathcal{A}g'_i$ at time t'_i. Naturally, there are usually different ways for the agents to solve a run instance and therefore we often want to evaluate the quality of a solution, which we denote by $qual(sol)$ and which is dependent on a particular instantiation of DTF.

B. The EIA

Without any idea when a task might be announced to the system it is very difficult to produce solutions of good quality (and sometimes even complete solutions at all). But for many problems, some knowledge can be collected that, if correctly used by the agents, can substantially improve the solutions A produces. One kind of such knowledge is about recurring tasks, i.e. tasks that in the past have been appearing very often in the run instances the system had to solve. The efficiency improvement advisor concept was developed as a general enhancement to A to identify such recurring tasks, determine what good solutions for these tasks would be, compare these solutions with the emergent solutions A produced and from this comparison, determine so-called exception rules. These exception rules are specific to individual agents and individual tasks, and are intended to result in a better (preferrably optimal) handling of these tasks in the future.

As presented in [9] and Figure 1, the EIA performs the following steps whenever it has gotten from all other agents all the information for a run instance: **receive** collects the local histories of all agents in A, **transform** creates out of the local histories a global history of the system, including information on the created emergent solutions, **extract** determines a sequence of recurring tasks from this global history, **optimize** solves the static optimization problem for this sequence, **derive** creates the above mentioned exception rules and **send** communicates those rules to the appropriate agents. For many of those steps there are different possible realizations, both in general and when applying the concept to a particular application area. While [9] introduced the so-called ignore exception rules that only told a particular agent not to take on a particular task (in the general form $cond_{ig}(s, d) \rightarrow \neg a_{ta}$, where s is the current situation an agent is in, d is its current internal state, a_{ta} is the action of starting a particular task ta and $cond_{ig}$ is the condition when the rule is triggered), [10] presented an additional kind of possible exception rules, so-called pro-active rules, that tell a particular agent to already prepare for a task at a particular point in time (in the general form $cond_{proa}(s, d) \rightarrow prep(ta)$, where $prep(ta)$ indicates the action(s) that an agent has to perform to prepare for performing ta).

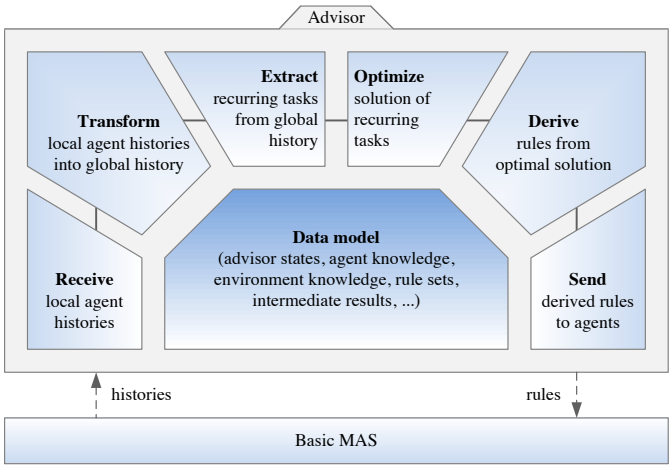

Fig. 1. EIA functional architecture

Since recurring tasks do not have to be in every run instance the system did in the past (not even in all the last k instances; how often a task needs to appear in those last k tasks to be deemed recurring is a parameter of the system), the knowledge about recurring tasks can be from time to time misleading (false knowledge of a task that will not appear in the current run instance). Another problem is naturally all the other tasks the system has to perform for which it does not have any information beforehand. Due to these problems, the EIA is supposed to be used when there is a high percentage of recurring tasks in every run instance and it has to be accepted that from time to time a particular run instance is solved worse than the solution would be without the EIA. But, as the experiments in the area of pickup-and-delivery problems in the two works from above showed, over several run instance there is always a gain with the advisor when there is a high percentage of recurring tasks.

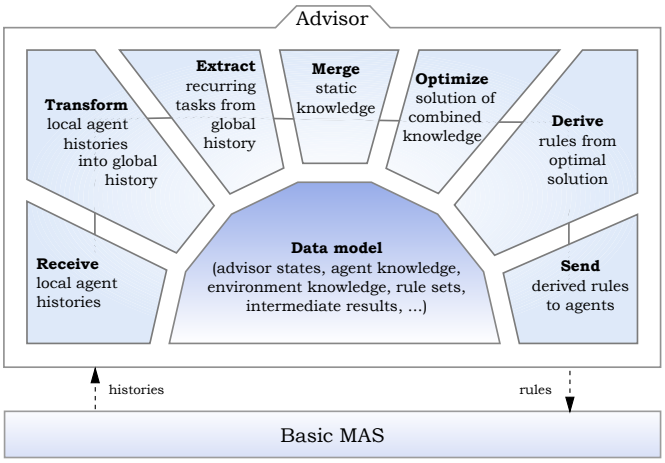

Fig. 2. EEIA functional architecture

C. The EEIA

As already mentioned, there are other sources of knowledge that the advisor can use to create exception rules. One such source are tasks that are known in advance of the whole interval $Time$. The EIA does not use this knowledge and such tasks are treated like the dynamically appearing tasks, which naturally can be very inefficient. The EIA can be extended to what we call the EEIA (*extended efficiency improvement advisor*) to also use knowledge about such known-ahead tasks, which requires sets of extension rules about every known-in-advance and every recurring task.

In general, the EEIA modifies the working cycle of the EIA by including a **merge** step after the **extract** step that merges the extracted recurring tasks with the known-ahead tasks and feeds this combined set into the **optimize** step (see Figure 2 and [7]). While the functionality of nearly all steps from the EIA stays the same for the EEIA, the **derive** step is substantially modified from what was done in [9], as mentioned above.

We will provide more details about EIA and EEIA when we discuss their application to the multi-hammer Whack-a-mole problem.

III. Multi-hammer Whack-a-mole

Most people know Whack-a-mole as a game at a carnival or an arcade. The (human) player is armed with a hammer and has to hit moles that appear seemingly randomly out of one of the holes arranged in a rectangle (the playing field or environment). After appearing, a mole stays out of its hole for a limited time or until it is hit by the hammer. The goal of the game is to hit as many of the moles as possible within a given time interval. In this form, the random appearance of the moles creates highly dynamic tasks for the player.

Having several players with hammers working the same playing field often adds to the fun due to the possibility of collisions between players when trying to get their hammers to a mole. While in todays world the randomness of the appearence of the moles would be based on the use of random

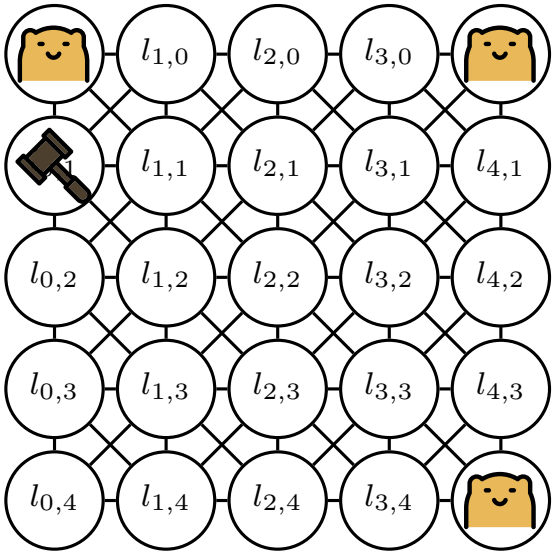

Fig. 3. A Whack-a-mole game situation

number generators that make it very difficult to predict when and where a mole might raise from its hole, the early days of this game required a mechanical solution for this with quite some possibility for making such predictions. And, naturally, having pre-announced mole appearences can create additional fun, especially if several players are involved (at least their collisions due to this are fun for the spectators).

All the above means that Whack-a-mole is an instantiation of dynamic task fulfillment. More formally, we can model the environment $\mathcal{E}nv$ as an $N \times M$ array (grid) with each node of the array representing a hole, whether a mole is out of the hole and also possible positions of a hammer (see Figure 3). A hammer, which is an element of A, can travel from a grid node vertically, horizontally and diagonally. For our experiments, we used the following rules: the game is performed in discrete time steps. In each time step, a mole might rise from a hole and will stay risen for a particular number of time steps if not hit. After that number of time steps it will lower itself into the hole and is not visible to a hammer. A hammer can either move from a node to an adjacent node (in all directions, including diagonally), can stay put, or can hit a visible mole on the same node. If a hammer performs the hit action without a mole being on the same node, this move is wasted. Two hammers cannot be on the same node. At each time step, the moles perform their moves first (in a given order) and then the hammers act (also in a given order).

A task ta is represented by a mole appearing at a particular time step t and grid position (x_{ta}, y_{ta}) and ta becomes unfulfilled at time step $t_{ta,end}$ if not hit by a hammer before that time step. A run instance for Whack-a-mole are all tasks appearing in a game that lasts the length of $Time$. If all

tasks are fulfilled (i.e. all appearing moles get hit before disappearing) then the emergent solution of the game could be exactly described as stated before, i.e. as a sequence of triples of a task, a hammer and the time the mole was hit by the hammer. We deal with the fact that a mole might "escape", by describing it as the task associated with the mole being "fulfilled" by a special agent $\mathcal{A}g_{nh}$ (no-$h$ammer) at the time the mole disappeared. And then we define the quality $qual$ of such a solution as the number of tasks in the solution that were not fulfilled by $\mathcal{A}g_{nh}$ (or, in other words, the number of moles that were hit).

There are several possibilities how hammers can do their decision making. In our experiments, the basic decision making of a hammer is rather simple. If the hammer is on a node with a mole out of the hole, then it hits this mole. Otherwise, it creates a list of all moles out of their holes and moves towards the mole that is nearest to its current position. If the list is empty, the hammer stays at its current position. Dealing with other hammers is also simple: if the hammer wants to move in a position with another hammer already on it, it stays at its current position until the move is possible.

IV. EIA and EEIA for Whack-a-mole

In this section we present the instantiations of the different steps in the EIA and EEIA to improve the cooperation of hammers for Whack-a-mole. We also show how to modify the decision procedure for these hammers to make use of the advice created by the two advisor variants.

The instantiations of the steps **receive** and **transform** for the EIA are very simple, since Whack-a-mole does not have any problems with regards to observing the environment. Therefore the EIA itself can observe when tasks are announced and fulfilled (or not). While we have the hammers communicate their actions and perceptions after each game (run instance) to the EIA, the EIA itself has already observed the emergent solution for the game as

$$((((x_{ta,1}, y_{ta,1}), t_{ta,1}), ha_1, t_1), ..., (((x_{ta,m}, y_{ta,m}), t_{ta,m}), ha_m, t_m))$$

with $ha_i \in A \cup \{\mathcal{A}g_{nh}\}$. In order to react to possibly changing recurring tasks, the EIA stores the solutions from the k last games (similar to what was done in [9]).

These k solutions are the input into the **extract** step. Similar to [9], we are using sequential leader clustering [6] to identify recurring tasks in those k solutions. For clustering, we need a similarity measure sim on tasks and a threshold value $clustthresh$. We defined the similarity of two tasks $((x, y), t)$ and $((x', y'), t')$ as $sim(((x, y), t), ((x', y'), t')) = dist((x, y), (x', y')) + |t - t'|$. Here, $dist$ is the smallest number of hammer moves between the two positions.

The clustering process goes through all tasks in the k solutions and if the similarity of the initial cluster element of any cluster to the new task is above the threshold the task is added to that cluster. Else, a new cluster with the task as initial cluster element is created. After that, each cluster that has at least $minocc$ elements represents a task that we consider recurring. The initial element of each such cluster is included in the sequence of recurring tasks (ta_1^{rec}, ..., ta_p^{rec}) (we order these tasks based on their $t_{ta_i^{rec}}$-values).

The sequence of recurring tasks is the input to the **optimize** step. The goal of this step is to produce the optimal solution opt^{rec} for the recurring tasks without any other tasks around, which is now a static optimization problem. In our system we have used a branch-and-bound based optimizer for this step, creating opt^{rec} = (($ta_1^{rec,*}$, ha_1^{rec}, t_1^{rec}), ..., ($ta_p^{rec,*}$, ha_p^{rec}, t_p^{rec})), where $\{ta_1^{rec,*}, ..., ta_p^{rec,*}\} = \{ta_1^{rec}, ..., ta_p^{rec}\}$ and $ha_i^{rec} \in A$. There was one problem regarding the optimization step, namely that the optimizer needs the time steps $t_{ta_i^{rec},end}$ for all i's, where the advisor was not able to observe when the corresponding mole disappeared, i.e. when a hammer was always fulfilling the task. In our implementation we chose to use as end time 8 time steps after a mole's appearance in such situations.

The idea of the EIA is to provide advice to agents about tasks that were not solved well by the whole system. In order to already evaluate how restrictive the advice scheme by the EEIA is, we use this scheme already for the EIA and as a result **derive** provides advice for each of the tasks in opt^{rec}. More precisely, for each of the (ta_i^{rec}, ha_i^{rec}, t_i^{rec}) in opt^{rec}, we create the following exception rules:

- For ha_i^{rec}, we create
  - a pro-active rule that has as trigger condition $cond_{proa}$
    - having reached the time $t_{ta,i}^{rec} - preptime(ta_i^{rec}) - clustervar(ta_i^{rec})$ and
    - having not reached the time $t_{ta,i}^{rec} + timeout$ and
    - ha_i^{rec} has not already performed $prep(ta_i^{rec})$ after the time $t_{ta,i}^{rec} - preptime(ta_i^{rec}) - clustervar(ta_i^{rec})$ and,
    - in case that ta_i^{rec} is not the first task assigned to ha_i^{rec} in this run instance and that ta' is the task ha_i^{rec} has to fulfill directly before ta_i^{rec} in opt^{rec}, having fulfilled ta'

    and as action $prep(ta_i^{rec})$.
  - an ignore rule that has as trigger condition $cond_{ig}$ that time $t_{ta,i}^{rec} - preptime(ta_i^{rec}) - clustervar(ta_i^{rec})$ has not been reached and as action $\neg a_{ta_i^{rec}}$.
- For every ha_j^{rec} with $i \neq j$ we create
  - an ignore rule that has as trigger condition $cond_{ig}$ that
    - fulfillment of task ta_i^{rec} can be started and
    - time $t_{ta,i}^{rec} + timeout$ has not been reached

    and as action $\neg a_{ta_i^{rec}}$.

Here, $preptime(ta)$ is a function that provides for a task ta the time that the hammer needs to prepare for hitting the mole represented by ta, which is the number of moves it has to make to get to the task's position plus one. The function $clustervar(ta)$ returns for a given recurring task ta the variance in starting times of the tasks within the cluster to ta. And $timeout$ is a system variable that determines how long a hammer waits for a task that it was assigned by a pro-active

rule to be performable before looking for other tasks to do. Also, remember that t_{ta} is the time when the mole represented by the task appears out from its hole.

In our realization of the EIA for Whack-a-mole, all exception rules a hammer is given are used in the following manner. If the condition of a pro-active rule is fulfilled, then the hammer will use only the associated task in his list of moles out of their holes (even if that mole is not out of its hole, yet), moving towards that particular position or staying at that position if it is already reached and hitting the mole the moment it gets out of the hole. The way how pro-active rules are created ensures that there is at most one pro-active rule with a fulfilled condition. If only the conditions of ignore rules are fulfilled, the hammer uses them to eliminate out of its list of currently visible moles all moles that represent tasks that an ignore rule targets. After that step, the hammer procedes as normal, selecting among the remaining tasks the one with the mole closest to the hammer and moving towards it. If the list is empty the hammer stays put.

The **send** step of the EIA happens before a new game is started and the sent out exception rules replace the exception rules a hammer had before.

As already stated, the EEIA uses the same realizations of the steps **receive**, **transform**, **extract**, **optimize** and **send** as the EIA. It adds the step **merge** after **extract**, which gets in addition to the recurring tasks a set of tasks $\{ta_1^{known},...,ta_q^{known}\}$, which are the known-ahead tasks for the next run instance. The EEIA checks if any of these known-ahead tasks are also in the recurring tasks, since there can only be one mole getting out of one hole, and if there are such tasks the duplicates will be eliminated and the remaining tasks will be the input for the **optimize** step.

As stated above, the Whack-a-mole instantiation of the **derive** step of the EIA creates sets of exception rules for each of the tasks that are optimized in **optimize** is identical to what the **derive** step of the EEIA creates (to allow us to see how much these rules influence the ability to deal with dynamically appearing tasks), so that the only difference in **derive** is that we are creating these rules for both the recurring tasks and the known-ahead tasks.

V. Experimental Evaluation

In this section, we will first discuss how we created the experiments we did, which required to both create system runs that allowed EIA and EEIA to be applicable and randomness to evaluate the generality of the concepts within the boundaries of their applicability. Then we will present the results of these experiments.

The EIA (and also the EEIA) requires that there are recurring tasks in a sequence of run instances (Whack-a-mole games), so that after they are identified exception rules can be created. But, in order to be realistic, we also need these recurring tasks to not occur in every game to explore the dangers advice can produce. And we naturally need dynamically occuring tasks in each run instance. To bring all these requirements together, we used the following procedure to create sequences of Whack-a-mole games, centered around the parameter δ (a similar procedure was used in [7]).

For run instances of length m in a sequence that form a run, we first create a so-called base instance $(ta_1, \ldots, ta_m)$ of random mole appearances, i.e. at random places $(x_{ta,}, y_{ta})$ in an $N \times M$ environment at random times t within $Time$ and with random disappearing times $t_{ta,end}$ after t_i (but within $Time$). If we create a mole appearance at a location we already have a task for, we create a new appearance, if the appearance time intervals overlap. Each of those ta_i is then assigned a probability p_i from the range $[\delta, 1.0]$ (distributed evenly over all ta_i) that will be used in the construction of all games within a sequence in a weighted coin toss to determine if ta_i will be in the game or not. More precisely, any game within the sequence is created as $\{ta'_1, ..., ta'_m\} \cup \{ta_1^{known}, ..., ta_q^{known}\}$, where $ta'_i = ta_i$ if a random number created between 0 and 1 is smaller or equal to p_i and ta'_i is a new randomly created mole appearance event else (with the same limitations as for the base instance). The tasks in the set $\{ta_1^{known}, ..., ta_q^{known}\}$ are also randomly created mole appearances that the EEIA will be informed of before a game starts. In all our experiments we used for this number q of known-ahead tasks $q = 5$.

An individual run consists of 55 run instances (games) created in the way described above. We used as the number of past games that EIA and EEIA look at for recurring tasks $k = 5$ and we consider a task reoccuring, if its cluster has a size $minocc = 4$. This means that depending on δ we have that tasks from the base instance might from time to time not even be considered a recurring task and if it is considered recurring it might still not occur in a particular game. The later means that EIA or EEIA will have a hammer preparing for that task, but it will never appear, which means that that part of the knowledge the advisor used was then misleading (and perhaps resulting in a worse game performance than the basic strategy). Naturally, this is always the danger when we use analytics of the past to predict the future. For the remaining EIA and EEIA parameters, we used $timeout = 10$ and, due to how we constructed the experiments, $clustervar(ta) = 0$ for every ta.

Table I presents the results of our experiments for various environment sizes (and consequent numbers of tasks) and two hammers for a δ-value of 0.85, which reflects the successful ratio of recurring tasks to dynamic tasks for the EIA from [9] and [10]. For each size and task number we performed 5 experiments. As can be seen, the EIA variant is for most sizes better than the basic strategy in more of the experiments than not, despite treating the known-ahead tasks as dynamic and despite creating more exception rules than what was originally suggested. In fact, if we look at the reported averages over all 5 experiments for a size, only for the 5×5 the average of the basic strategy is better than the EIA average. This means that the new scheme in the **derive** step is well targeted and not too restrictive to deal with dynamic tasks.

The EEIA, due to treating the known-ahead tasks as part of the planned for tasks, is nearly always better than the EIA (the exceptions being the second experiment for 3×5 and the forth

TABLE I
COMPARISON BASE SYSTEM, EIA AND EEIA FOR $\delta = 0.85$

$N \times M$	m	basic strat.	EIA	$EEIA$
3×5	15	251	236	277
3×5	15	249	265	255
3×5	15	251	248	265
3×5	15	264	256	264
3×5	15	239	270	260
avg.		250.8	255.0	264.2
4×5	20	266	296	314
4×5	20	298	302	333
4×5	20	320	309	331
4×5	20	299	300	323
4×5	20	325	306	329
avg.		301.6	302.6	326.0
5×5	25	349	341	386
5×5	25	352	361	395
5×5	25	361	338	386
5×5	25	364	347	377
5×5	25	336	338	371
avg.		352.4	345.0	383.0
6×5	30	410	380	438
6×5	30	355	366	396
6×5	30	345	380	414
6×5	30	328	373	401
6×5	30	317	368	405
avg.		351.0	373.4	410.8
7×5	35	396	423	442
7×5	35	355	407	435
7×5	35	342	385	427
7×5	35	382	377	373
7×5	35	329	353	383
avg.		360.8	389.0	412.0

TABLE II
COMPARISON BASE SYSTEM, EIA AND EEIA FOR $\delta = 0.75$

$N \times M$	m	basic strat.	EIA	$EEIA$
3×5	15	260	228	265
3×5	15	237	259	268
3×5	15	264	248	269
3×5	15	251	254	268
3×5	15	259	239	248
avg.		254.2	245.6	263.6
4×5	20	296	283	316
4×5	20	293	301	309
4×5	20	313	307	310
4×5	20	286	294	304
4×5	20	303	298	313
avg.		298.2	286.0	310.4
5×5	25	283	313	353
5×5	25	328	317	377
5×5	25	363	333	353
5×5	25	303	293	316
5×5	25	368	345	368
avg.		329.0	320.2	353.4
6×5	30	391	376	412
6×5	30	349	324	359
6×5	30	363	379	409
6×5	30	345	357	387
6×5	30	322	318	358
avg.		354.0	350.8	385.0
7×5	35	347	368	393
7×5	35	425	380	429
7×5	35	388	388	405
7×5	35	378	412	417
7×5	35	401	372	395
avg.		387.8	384.0	407.8

experiment for 7×5) and only for one experiment (again, the forth experiment for 7×5) worse than the basic strategy. If we look at the averages, then the EEIA is always better than the basic strategy and the EIA, with the differences increasing with larger environments and numbers of tasks.

Table II highlights the already mentioned danger of using analytics when they produce misleading knowledge. For two hammers and with a δ-value of 0.75, there will be in nearly every game for the EIA supposedly recurring tasks that the EIA plans for that will not materialize in the game for which the plan was made. And this often leads to kind of freezing a hammer for some time, which naturally is not happening with the basic strategy. Consequently, for many of the experiments we have the basic startegy being better than the EIA, although for each of the environments we have at least one experiment where the EIA is better, But, if we look at the averages, the basic strategy is always better, even if the differences are getting smaller with larger environments and numbers of tasks.

The EEIA, on the other hand, was able to use the additional sure knowledge about the known-ahead tasks to not only outperform the EIA, but also the basic strategy, when looking at the averages. For most of the environments we have one experiemnt where the basic startegy is better than the EEIA, but at most one of those. And in all experiments the EEIA is better than the EIA.

In Figure 4 we take a closer look at one of the runs in the 7×5 environment for $\delta = 0.85$. The two graphs depict the performance of the EIA (top) and EEIA (bottom) compared to the basic strategy (represented by the 0 line). The graphs start with the 5th run instance, since this is the first run instance both advisor variants can provide advice (given how we generated the runs).

Both graphs show that much more run instances are above the 0 line than below, although for the EEIA obviously less are below than for the EIA (4 vs. 7). For the EIA the largest improvement is 6 hits (in 6 run instances) whereas the EEIA goes up to 13 more hits than the basic strategy (requiring different scales for the two graphs). Both graphs show the effects of dynamism leading already to the basic strategy being for some run instances better (i.e. nearer the optimum) and for some worse (allowing for more or less improvement by the advisors). And, naturally, the EIA has a higher volatility due to having 5 more dynamic tasks to deal with than the EEIA. Due to using the knowledge gained from analyzing the history (and our experimental setup that will make this knowledge misleading for some run instances) we have to accept from time to time behavior that makes the adviced hammers worse than their basic behavior, but the gains in other run instances make more than up for this (as both the graphs and Table I show).

VI. DISCUSSION OF GENERALITY OF THE EIA AND THE EEIA

As stated in the introduction, the goal of this paper is to show that simple games (with slight variations) can be used to strengthen generality claims made for AI and MAS concepts.

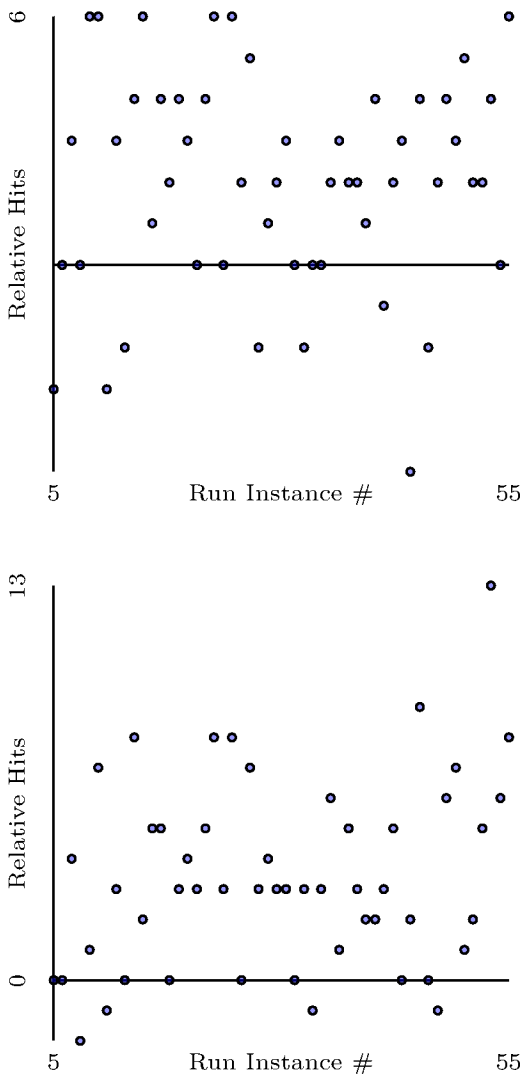

Fig. 4. Looking at a single run (of EIA on top, of EEIA on bottom)

For EIA and EEIA, the initial, complex application was to pickup-and-delivery problems (see [2]), where the tasks to fulfill were a truck picking up some goods at one place and delivering it to another place. A run instance consisted of all tasks a group of trucks had to do in a day and a sequence of run instances obviously covers the tasks over a certain number of days.

The evaluation of the EIA and the EEIA based systems for the pickup-and-delivery problem in [7] is similar to what we have done in the last section. A summary of the results from [7] is as follows: For $\delta = 0.75$ and for smaller numbers of tasks in a run instance, on average the basic system was better than the EIA, but for larger numbers of tasks the EIA was better. The EEIA was better than both the basic systems and the EIA and the improvement by the EEIA was growing with larger numbers of tasks. For $\delta = 0.85$, both EIA and EEIA were better than the basic system and the EEIA was consistently better than the EIA.

As we have seen in the last section, the results for EIA and EEIA for multi-hammer Whack-a-mole are similar to these results from [7], which allows us to claim with some generality that EIA and EEIA are indeed improvements for the behavior of self-organizing multi-agent systems if their application requirements are fulfilled.

VII. Related Work

There is a long tradition of using games to evaluate concepts in multi-agent systems, like the iterated prisoner's dilemma [1] or variants of pursuit games [4]. But, naturally, evaluating concepts on applications that are of relevance in the real world, like transportation (as in [9]) or elevator control (as in [11]), shows better the practical relevance of such concepts, although, as already stated, the generality of the concept is also of quite some interest.

Whack-a-mole, itself, has been given a serious formal treatment. [5] contains proofs first that, in the general case, this is a computationally hard problem. It also provides a thorough competitive analysis, confirming that any dynamic strategy is generally not going to provide performance on par with the optimal. Although that is not information that should surprise anyone, [5] also analyzes further special cases. These include special formulations of what we call run instances, where the dynamic strategies perform well and where optimal can be calculated in polynomial time. It alternately gives special cases which appear to have these special properties, but deceptively are as difficult as the general case. The majority of their analysis is done general to "deterministic dynamic" players, so applies to the hammer agent we have discussed.

Looking at generality claims around multi-agent system approaches for dynamic optimization problems, [11] comes nearest to the EIA/EEIA concept, but does not really make any generality claim except for some common terminology (i.e. Controller/Observer architecture). While [11] presents 3 applications, the solutions to these applications are rather different leaving the reader with the question which of those approaches would be best for their particular problem.

VIII. Conclusion

In this paper, we presented the simple game of Whack-a-mole as an example of a highly dynamic problem that can be used to evaluate multi-agent concepts that aim to improve the agents' behavior in solving dynamic problems. We applied the efficiency improvement advisor concept to Whack-a-mole getting similar results as for the previous application area of pickup and delivery problems. We also used Whack-a-mole to evaluate an enhancement of the EIA, the EEIA that uses additional knowledge about mole appearances made known before a game starts, which naturally results in more and better knowledge for the advisor. And, indeed, in the experiments, the EEIA performed substantially better than the EIA treating the known-ahead appearances as dynamic events. This shows that

variants of relatively simple games can be used to substantiate generality claims of concepts.

REFERENCES

[1] R. Axelrod: The evolution of strategies in the iterated prisoner's dilemma, in The Dynamics of Norms, Cristina Bicchieri et al. (eds). Cambridge University Press, 1997, 1–16.

[2] G. Berbeglia, J-F. Cordeau, G. Laporte: Dynamic pickup and delivery problems, European journal of operational research 202.1, 2010, pp. 8–15.

[3] M. Campbell, A.J. Hoane Jr., F.-h. Hsu: Deep Blue. Artificial Intelligence 134(1-2), 2002, 57–83.

[4] J. Denzinger and M. Fuchs: Experiments in Learning Prototypical Situations for Variants of the Pursuit Game. In Proc. ICMAS-96, Kyoto,1996, 48–55.

[5] S. Gutiérrez, S.O. Krumke, N. Megow, T. Vredeveld: How to whack moles, Theoretical computer science, 2006, pp. 329–341.

[6] J. A. Hartigan: Clustering Algorithms, John Wiley & Sons, 1975.

[7] N. Nygren and J. Denzinger: Extending the Advisor Concept to Deal with Known-Ahead Transportation Tasks. In Proc. IDCS 2018, Tokyo, 2018, 27–38.

[8] D. Silver, A. Huang, C.J. Maddison, A. Guez, A. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis: Mastering the game of Go with deep neural networks and tree search, Nature 529, 2016, pp. 484–489.

[9] J-P. Steghöfer, J. Denzinger, H. Kasinger, B. Bauer: Improving the Efficiency of Self-Organizing Emergent Systems by an Advisor, Proc. EASe 2010, Oxford, 2010, pp. 63–72.

[10] T. Steiner, J. Denzinger, H. Kasinger, B. Bauer, Pro-active Advice to Improve the Efficiency of Self-Organizing Emergent Systems, Proc. EASe 2011, Las Vegas, 2011, pp. 97–106.

[11] S. Tomforde, H. Prothmann, J. Branke, J. Hähner, M. Mnif, C. Müller-Schloer, U. Richter, H. Schmeck: Observation and control of organic systems, In Organic Computing – A Paradigm Shift for Complex Systems, Springer, 2011, pp. 325–338.