THE UMA-SOM PLATFORM FOR GENERAL LEARNING

DAN P. GURALNIK† AND DANIEL E. KODITSCHEK†

A technical report on work executed under
AFRL SIRCUS grant FA865015D1845 (subcontract 6697371)
01/2016–12/2017

Abstract. A partially distributed, GPU-accelerated implementation of a modality-agnostic architecture for life-long general learning, based on the Universal Memory Architecture (UMA) memory model previously introduced by the authors was developed, and is in the process of being tested in a range of distinct settings. The design emphasizes modularity, distributability and flexibility of application by separating the computational components of the architecture (implemented in CUDA) from the RESTful communication framework (C++) and the simulation state update and environment (Python).

We report on the usage, operation and capabilities of the current design, and review some experimental results to illustrate learning and scaling capabilities. This report is supplemented by a detailed technical report and user's manual which will be posted on the project's private GitHub repository, `https://github.com/kotmasha/kodlab-uma/docs/`, together with up-to-date distributions of the architecture (linux Ubuntu and Windows operating systems) and the simulation environments we have developed for it.

1. Overview

In preliminary work [3, 4] we proposed the use of the Universal Memory Architecture (UMA) as a modality- and context-agnostic tool for effecting simultaneous life-long learning and reactive control, potentially bridging between low-level, sub-symbolic and symbolic information processing, required for situated/embodied general agency.

In contrast with other representation/learning methods in AI/ML (e.g. optimal policy learning for POMDPs), UMAs are not used for representing a mapping from a state space to a set of available actions (an optimal policy), but, rather, are used for summarizing the interactions between the agent's experiences in a geometric structure—a *model space* $\mathbb{M}$,— whose geodesics serve as guides for reactive decision making. The mathematical framework underlying UMAs provides guarantees of efficient computation, putting low degree polynomial complexity caps on both the time complexity (cubic in the number of sensory bits) and space complexity (quadratic) of all maintenance and planning operations.

Unfortunately, the low complexity of maintenance and planning comes at the price of inaccuracies in the representation (a "no free lunch" phenomenon). However, all such inaccuracies are formally interpreted as obstacles to navigation along geodesics in the model space. Consequently, rather than attempt learning a unified UMA representation, encompassing all possible actions, by a single situated agent, we were led to maintaining *multiple* binary 'specialist' agents, each with its own UMA representation; each charged with deciding on whether to engage or not in a single, dedicated action. The distributed nature of this new setting is fertile ground for conflicts, for example: a set of specialists may each independently prefer to act despite that particular combination of actions being impossible or ineffectual in the context of the current state. Thus, within each decision cycle, the community of specialists will need to negotiate its response to the current collection of sensory stimuli given every agent's target experience.

The focus of the last two years of UMA research under the SIRCUS grant has been on the following:

- Formalizing the notion of a binary UMA agent (BUA)—our notion of a specialist—and its learning functionalities;

†University of Pennsylvania, Electrical & Systems Engineering Department

- Constructing a GPU-accelerated implementation of BUAs (UMA-SOM);

- Constructing a RESTful architecture to enable the maintenance of a situated community of BUAs capable of autonomous modification—or *bootstrapping*—in two forms:
  - **Sensory enrichment of individual BUAs:** Each BUA comes equipped with an initial collection of binary queries—its *sensorium*—which, as time progresses, may need to be updated with derived queries, to improve the predictive power of the BUA.
  - **Construction and merging of BUAs:** We contend that, given a task, useful patterns of negotiation between the initial population of BUAs could be realized through introducing additional BUAs which act by altering decisions made by existing peers.

- Constructing a collection of test environments and gathering performance statistics.

The bootstrapping operations we plan to test using the UMA-SOM system[1] are designed to leverage ideas from Minsky's "Society of Mind" work [5, 6] for the purpose of improving control of a situated agent by its BUA sub-agents as a result of emergent deliberation among them. As a result, the single greatest design challenge for the UMA-SOM architecture—in addition to the derivation of accompanying theoretical guarantees—has been to design and implement software ensuring that: (1) new BUAs could be introduced in response to internal deliberation by existing BUAs; (2) existing BUAs may be coalesced based, for example, on commonalities of behavior; and (3) individual BUAs are capable of extending their original set of queries with task-relevant derived queries.

This report on our progress is organized as follows: Section 2 provides an overview of the architecture and its current configuration; more details of the design, its capabilities and envisioned future development are provided in Section 3; available simulation environments and experimental results are reviewed in Section 4.

2. Products

We have developed a partially distributed software package capable of maintaining a situated community of BUAs and allowing nearly arbitrary interactions among them, facilitating experimentation with different bootstrapping operations (aimed at improving task-oriented control). The community is situated in the following sense: every BUA accepts a number of binary inputs from the computational environment—the *experiment*— where each input corresponds to (1) physical sensing in a real environment; (2) readings from a canned data set; and/or (3) function values reflecting the state of a simulated environment. The main design challenge has been to implement the necessary data structures in a way that makes each BUA agnostic of the source/semantics of its individual input streams.

A by-product of this effort is a conjectured method for bootstrapping emergent synchronization among BUAs (via the formation of 'arbiter' BUAs in run time), based on a single bootstrapping operation we have proposed earlier for the purpose of realizing Minsky's notion of a *censor* (which we will refer to as a *critic*).

2.1. Components of the UMA-SOM platform.

The UMA-SOM architecture assumes a situated agent interacting with its environment, on a discrete linear time scale (which is modeled by the natural numbers), by deciding, at every moment t in time, whether or not to engage in each of a fixed finite set $\mathbf{A}$ of *basic actions/behaviors*, every one of which spans the time interval from t to $t+1$.

In order to render the decision, the agent takes a complete reading from a collection $\mathbf{\Sigma}$ of *binary queries* to the environment—the *raw observation* at time t. $\mathbf{\Sigma}$ is always closed under complementation, which is henceforth denoted by $a \mapsto a^*$.

In other words, the environment may be treated as a "black box", receiving, at each time t, an input of a subset $D \subset \mathbf{A}$—the actions to be taken at that time—and returning the raw observation resulting from these actions as its output.

[1] Also see `https://kodlab.seas.upenn.edu/group/dang/umca/` for a brief annotated review of the theory underlying UMA-based agents and their development.

2.1.1. *Python front end (FE) managing the environment.* UMA-SOM enables the use of a dedicated Python script for initializing a collection of BUAs, one for each action α from $\mathbf{A}$; each with its prescribed collection $\mathbf{\Sigma}_\alpha \subseteq \mathbf{\Sigma}$ of initial queries (some or all queries may be shared by the different BUAs); each with its own *motivational signal* φ_α, provided as a function of the system's state. While, in some contexts it is correct to interpret φ_α as a *reward signal*, a BUA need not always be assigned the task of maximizing reward; the role of φ_α is more precisely seen as that of influencing the BUAs decisions regarding what observations should be ignored as part of the process of deriving its dynamic implications record. In the current implementation, due to the method used for learning implications, φ_α is a variable of type float, taking values in $[1, \infty)$.

The simplest use case is that of connecting a "real-world" system, such as a robot, to be treated by the UMA-SOM system as a black box in the manner described above. The only restriction one faces in this setting is imposed by the time it takes the BUAs to perform their deliberations and report a decision. At the other end of the spectrum, complete simulations present the most complex use cases, if one uses the UMA-SOM libraries to write a complete stand-alone simulation detailing the interactions among the environmental variables, the queries available to each BUA, and the actions represented by each BUA.

We leverage the weak typing in Python to enable run-time alterations to all components, for example: motivational signals of individual and the effects of actions may be altered in run-time.

However, this capability imposes a weakness on the current implementation, as the front end is responsible for maintaining a record of any queries introduced in run time. This places the burden of maintaining an ever-growing global record on the same CPU thread that updates the experiment state (the BUAs states are updated separately). We plan to address this weakness in the future by encapsulating each BUA in its own 'experiment', to enable a more distributed/layered state update, as well as an asynchronous approach to UMA-SOM, letting the system benefit from the lower latency of the decision cycle of leaner BUAs.

2.1.2. *C++/CUDA implementation of a single BUA.* The back end (BE) implements each BUA α using the following data structures. Two *snapshot* objects are maintained, labelled with α and α^*, both accepting the input of (1) a collection $\mathbf{\Sigma}_\alpha$ of $2n_\alpha$ initial Boolean inputs (counting complements), which we refer to as the BUAs initial *queries* or *sensors*, and (2) an up-to-date value of the motivational signal, denoted[2] by $\varphi_\alpha|_t$.

Dynamically Extended Sensorium (maintained by FE on CPU): At any time $t \geq 0$, each snapshot $\beta \in \{\alpha, \alpha^*\}$ has associated with it a collection $\mathbf{\Sigma}^\beta|_t$ of Boolean queries[3], of size $2N^\beta|_t \geq 2n_\alpha$, including the initial $2n_\alpha$ input queries, with the additional queries, if any[4], being *delayed conjunctions* of the form:

$$s = \sharp(\bigwedge_{a \in A} a)\,, \qquad A \subset \mathbf{\Sigma}_\alpha \tag{1}$$

and their complements, where $\sharp$ is the 1-step delay operator[5].

Throughout this section, we will omit the time index wherever possible, for the sake of legibility. At a high level, each snapshot $\beta \in \{\alpha, \alpha^*\}$ has the following attribute fields:

Weight Matrix (maintained by BE on GPU): A dense[6] floating point symmetric matrix $\mathbf{W}^\beta = [w^\beta_{ab}]_{a,b \in \mathbf{\Sigma}^\beta}$ of dimensions $2N^\beta \times 2N^\beta$, where, loosely speaking, w^α_{ab} represents the significance[7] ascribed by the BUA to the event $a \wedge b$ given α is active; and, respectively, $w^{\alpha^*}_{ab}$ stands for the significance of $a \wedge b$ given α is inactive.

Implications Matrix (BE,GPU): A Boolean $*$-symmetric[8] matrix $\mathbf{D}^\beta = [D^\beta_{ab}]_{a,b \in \mathbf{\Sigma}^\beta}$ of dimensions $2N^\beta \times 2N^\beta$, where $D^\beta_{ab} = \texttt{True}$ means a is believed to imply b given β (in particular, $D^\beta_{aa} = \texttt{True}$ and $D^\beta_{aa^*} = \texttt{False}$ always). These form the basis for inference and planning.

---

[2]Henceforth to be read as: "phi sub alpha at time t".

[3]Hereafter, we adopt the following indexing convention: initial parameters of a BUA α all have a *subscript* α, but all objects/parameters associated with the snapshots $\beta \in \{\alpha, \alpha^*\}$ carry a *superscript* index of β.

[4]For example, pursuant to the above indexing convention, at time $t = 0$ we have $\mathbf{\Sigma}^\alpha = \mathbf{\Sigma}^{\alpha^*} = \mathbf{\Sigma}_\alpha$ for every BUA α.

[5]That is, $\sharp q$ at time t coincides with q at time $t - 1$ for any query q.

[6]An $m \times n$ matrix is said to be in *dense representation* when it is given as a two-dimensional array, $((a_{ij})_{j=1}^n)_{i=1}^m$); a *sparse* representation of the same matrix takes the form of a dictionary $\{ij : a_{ij} \mid a_{ij} \neq 0\}$.

[7]Here, significance is derived from the motivational signal, see Equation (2) below.

[8]Whenever applicable, a matrix $M = [m_{ab}]$ is $*$-symmetric if $M_{ab} = M_{b^*a^*}$ for all $a, b \in \mathbf{\Sigma}$.

Delay Mask (BE,GPU): A Boolean matrix $\texttt{Mask}^\beta$ of size $2n_\alpha \times 2(N^\beta - n_\alpha)$ encoding the composition of the delayed queries $\sharp(\bigwedge_i a_i) \in \mathbf{\Sigma}^\beta$ in $2n$-dimensional Boolean vectors (each coordinate corresponding to an initial query).

Additional data regarding the state of the internal deliberation is stored in *signals*, which are Boolean $2N^\beta$-dimensional vectors indexed by Σ^β:

Raw Observation Signal (FE broadcasts to BE,GPU): $\texttt{Obs}^\beta$ stores the current values of the input bits from the experiment.

Current State Signal (BE,GPU broadcasts to FE): $\texttt{Curr}^\beta$ represents the BUA's belief regarding the current state, obtained by reconciling $\texttt{Obs}^\beta$ with the implication component $\mathbf{D}^\beta$ of the BUA's belief state.

Prediction Signal (BE,GPU broadcasts to FE): $\texttt{Pred}^\beta$ represents the subspace of $\mathbb{M}^\beta$ predicted to contain the next state.

Target Signal (BE,GPU; or set by FE): $\texttt{Targ}^\beta$ represents the inferred target subspace of the model space $\mathbb{M}^\beta$.

In general, a signal should be thought of as encoding a (possibly empty) convex subset of the model space $\mathbb{M}^\beta$ associated with the snapshot. It is a fundamental property of such model spaces, that every convex set of states is characterized by the set of queries witnessed by its points (see [4], Section 3.5).

Skipping computational details, the decision cycle of a BUA employs the data structures above in several stages, as follows:

(1) **Raw observation (FE,CPU and BE,GPU).** The raw observation $\texttt{Obs}^\beta|_t$ is computed from initial sensor values broadcast by the experiment and from stored previous raw observation values.

(2) **Weight Update (BE,GPU).** The currently implemented weight update is a completely parallel conditional discounted integrator of the motivational signal:

$$w^\beta_{ab}|_t := \begin{cases} q_\alpha \cdot w^\beta_{ab}|_{t-1} + (1-q_\alpha)\varphi_\alpha|_t \cdot \texttt{Obs}^\beta_a|_t \texttt{Obs}^\beta_b|_t & \text{if } \beta \text{ is active} \\ w^\beta_{ab}|_{t-1} & \text{if } \beta \text{ is inactive}, \end{cases} \tag{2}$$

where q_α is a parameter in the range $(\frac{1}{2}, 1)$, assigned to the BUA at 'birth' (but could be altered in run time).

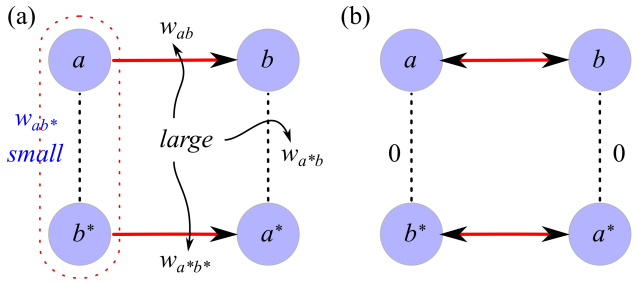

FIGURE 1. Illustrating a general principle for deriving a directed edge (a) and a bidirectional edge (b) in the poc graph corresponding to the weight matrix $\mathbf{W}^\beta$ (specifics in [4], Sections 4.2-4.4).

(3) **Implications Update for the active snapshot (BE,GPU).** A separate CUDA kernel uses the matrix $\mathbf{W}^\beta$ to construct a Boolean matrix $\mathbf{G}^\beta$ representing the derived *poc graph* (see Figure 1), a directed graph whose transitive closure is the approximate implication order on $\mathbf{\Sigma}^\beta$, represented by the matrix $\mathbf{D}^\beta$, which is then computed from $\mathbf{G}^\beta$ using a GPU implementation of an appropriately modified Floyd's algorithm [2] (time complexity cubic in N^β, but severely reduced due to the use of a GPU). Future implementations will take advantage of the sparsity of these matrices to further improve performance.

(4) **Current State Update (BE,GPU).** An operation called *coherent projection* is applied to the raw observation $\texttt{Obs}^\beta|_t$ in order to reconcile it with the BUA's belief state ([4], Section 3.6). The result is stored in $\texttt{Curr}^\beta|_t$

(5) **Enrichment and Pruning (FE+BE).** The BUA may introduce or remove an arbitrary number of delayed conjunctions—recall Equation (1)—to the currently active snapshot at this point. The specific protocol is currently governed by the Python FE—to make tweaks easier to introduce—and relies heavily on propagation (see below). The goal of the process is for $\mathbf{\Sigma}^\beta$ to eventually stabilize in a state wherein the BUA's prediction encoded by $\texttt{Pred}^\beta|_t$ always falls within the perceived state, encoded by $\texttt{Curr}^\beta|_t$, while keeping N^β as small as possible. Once $\texttt{Curr}^\beta|_t$ has been made available, a comparison with $\texttt{Pred}^\beta|_t$ facilitates the computation of $\mathbf{\Sigma}^\beta|_{t+1}$. This computation is currently partially delegated to the front end, to enable experimentation with different procedures.

(6) **Computing a target state (BE,GPU).** Current implementations allow for maintaining either a target specification directly supplied by the user (via the front end), or an intrinsic target derived from the weight matrix. Written as a subset of $\mathbf{\Sigma}^\beta$, the target signal *for time* $t+1$ is currently computed as:

$$\texttt{Targ}^\beta|_{t+1} = \left\{ a \in \mathbf{\Sigma}^\beta|_{t+1} \;\middle|\; w^\beta_{aa}|_t > w^\beta_{a^* a^*}|_t + \epsilon_a \right\}, \tag{3}$$

where $\epsilon_a \geq 0$ is a free parameter (currently set to 0). *This target specification is guaranteed to encode a non-empty (convex) domain in the model space* $\mathbb{M}^\beta$. In a nutshell, this target specification provides the agent with a notion of "median position" in the model space, with respect to the most recently experienced values of the motivational signal; experiencing a relatively high value of this signal in the context of a sensation $a \in \mathbf{\Sigma}^\beta$ results in that sensation becoming desirable for a while (depending on the value of the learning coefficient q).

(7) **Deliberation and Decision (BE,GPU).** In order to form the prediction $\texttt{Pred}^\beta|_{t+1}$ of the next state, the mask $\texttt{Mask}^\beta|_t$ is used to extract from the current state $\texttt{Curr}^\beta|_t$ predicted values for the delayed queries. Propagation is then applied to these partial signals to complete the prediction signals $\texttt{Pred}^\beta|_{t+1}$. Each snapshot then computes a *divergence* $\delta^\beta := \texttt{Div}\left(\texttt{Pred}^\beta|_{t+1}, \texttt{Targ}^\beta|_{t+1}\right)$, and the agent picks action (broadcasts α) if $\delta^+ < \delta^-$; inaction if $\delta^- < \delta^+$ (broadcasts α^*); and breaks the tie randomly otherwise.

Propagation. An operation called propagation (see [4], Section 5.2.2, for details) underlies stages (4)–(7). Its geometric meaning is that of nearest point projection in the model space $\mathbb{M}^\beta$. Its complexity is dominated by matrix multiplication over the $(\vee, \wedge)$-algebra, of the matrix $\mathbf{D}^\beta$ by batches of various input signals.

2.2. **Unit Testing Capabilities.** Two additional unit testing applications were developed to guarantee the proper operation of separate BE components (kernel functions, snapshot maintenance) and of the RESTful communications framework, using Google Test[9].

3. Design and Limitations

In this section we describe the design and operation of the UMA-SOM architecture. A high-level schematic is provided in Figure 2.

3.1. **Python Front End.** The FE is charged with maintaining the state of an experiment and executing an orderly state update.

3.1.1. *Components.* The user provides a Python script with instructions for constructing an `Experiment` object, whose purpose is to keep track of the state of the desired experiment/simulation:

(1) The experiment object maintains an ordered list of *measurables*, serving as components of the state of the experiment. Each measurable is equipped with:

Immutable unique identifier (ID): Either provided by the user, or constructed by the FE, IDs are used for ordering the state update process. They also provide the user with flexibility to define arbitrary interactions among the measurables.

[9] `https://github.com/google/googletest`

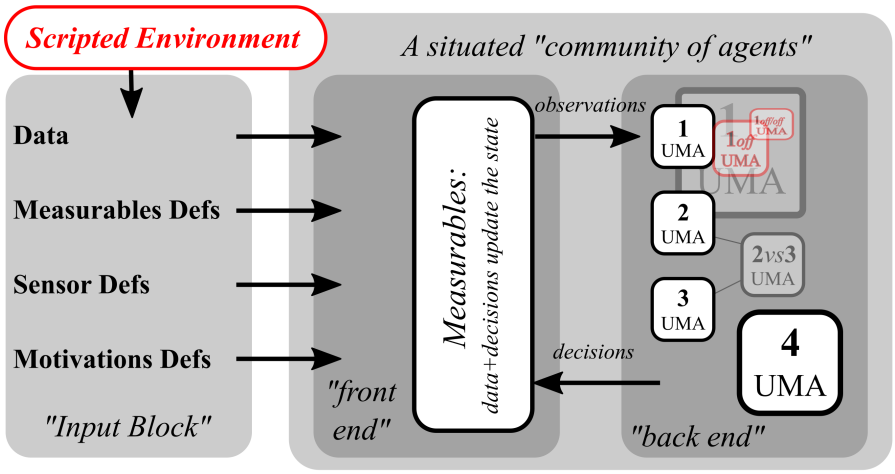

FIGURE 2. The current design of the UMA-SOM architecture. The transparent UMA blocks depicted in the back end signify the potential of BUAs to evolve a larger sensorium, as well as to spawn other BUAs to facilitate communal deliberation.

State/history vector: An ordered list[10] containing a prescribed number of latest values of the measurable, starting with the current value;

Update function: An instruction on how to update the state vector based on the history of the experiment and the currently known decisions made by the BUAs in the experiment (see below).

(2) Every experiment object is initialized with a privileged *decision measurable*, identified as `dec`, whose purpose is to accumulate the decisions of the BUAs present in the experiment over the course of each run of the update process.

(3) The user-provided script operates by extending the list of measurables:

Registration: The user assigns identifiers to measurables/BUAs ahead of construction, in the order in which their values are to be computed during the update process (see Section 3.1.2 below).

Construction: Constructing a measurable amounts to providing its initial value and its update function. Boolean measurables intended as queries for the use of BUAs are introduced in complementary pairs and are referred to as *sensors*. Finally, calling the constructor for an `Agent` with ID α implements a BUA with a prescribed motivational signal (necessarily a registered measurable) and the associated data structures in the BE, and adds a complementary pair of sensors with IDs α, α^*, to house the value of the BUAs action, when it is taken.

Simulation: The user specifies the desired simulation loop (e.g., the desired number of calls to the `experiment`'s `update_state` function; how to process the experiment state to form output).

3.1.2. *Operation.* Each update cycle is split into two *sweeps*.

Decision Sweep: Going over the ordered list of IDs, one focuses on (1) the present BUAs, and on (2) the measurables marked by the user as participating in the deliberation process (e.g. any state variables dependent on the internal deliberation of BUAs appearing earlier in the sweep and whose values are required by BUAs which appear later in the sweep). For each relevant ID, one executes the following:

(1) At the start of the sweep, `dec` is reset to equal the empty list.

(2) Each BUA α acts by depositing its identifier α (decision to act), or its complement α^* (decision not to act), in the `dec` measurable, but the values of the *measurables* α, α^* are *not* updated.

[10]To be more precise, a Python `deque` of prescribed length.

(3) Each non-BUA measurable marked for participation in this sweep is updated using its update function.

Response Sweep: At this point, all BUAs have deposited a token identifier in the `dec` measurable, so the experimental environment is ready to be updated. Going over all identifiers, the values of measurables skipped in the decision sweep (this includes all BUAs!) are updated using their update functions, while measurables which participated in the decision sweep are skipped.

3.1.3. *Generic Use Cases for Measurables.* While the `Experiment` class is designed to accommodate arbitrary discrete time simulation environments, the following types of measurables turned out as either essential or, at least, ubiquitously useful:

Sensors for BUAs: Boolean 'Sensors' are a necessary class of measurables providing the substrate for the operation of BUAs. They are always constructed in complementary pairs, each with a state vector of default length 2, to accommodate the current state and the preceding state, facilitating the autonomous construction of delayed conjunctions. The user is free to assign any subset of the initial collection of sensors to any of the different initial BUAs.

BUA-associated sensors and deliberation: The constructor of each BUA α introduces the complementary sensor pair (with IDs α, α^*). By the time the response sweep is completed, the values of these sensors represent the actual BUA state ('active'/'passive'). The update function for this sensor, which may be modified in run time, reflects the desired deliberation process among agents as follows. The initial decision by α is recorded by the BUA depositing α ('act') or α^* ('do not act') in the `dec` measurable; then the value of the update function for α may coincide with the Boolean value of the expression $\alpha \in$ `dec` (meaning the BUA α is unaffected by external deliberation), but could also be set to a function of additional measurables, including decisions made by other agents.

Randomization: Randomizing the state of the experiment is made easy through introducing measurables whose value is a random number generated by any of the available Python libraries. This kind of measurable may be applied, for example, to impose hard-wired arbitration between a pair of conflicting BUAs α, β in a situation when $\alpha \wedge \beta$ is not physically possible: if both α and β decide to act, the user may introduce a Bernoulli-$\frac{1}{2}$ measurable `flip`, setting the update to $\alpha \leftarrow (\alpha \in$ `dec`$) \wedge$ `flip` and $\beta \leftarrow (\beta \in$ `dec`$) \wedge \neg$`flip` in the event that $(\alpha \in$ `dec`$) \wedge (\beta \in$ `dec`$)$.

External Input: Any external information, such as values from sensors mounted on a robot, could be introduced through a measurable whose value is a constant pointer to a fixed Python object in the initialization script, in charge of communicating the values of all BUA sensors to the robot (motor instructions) and collecting the sensory data produced in response to the robot executing those instructions.

3.1.4. *Front End Design Limitations.* Intuitively, the decision sweep allows each BUA to declare its *intention* for the next state, leaving each agent's update function available for determining *how exactly* to act on its intention, given the decisions of other agents and any values of measurables computed so far. At the same time, however, the inflexible ordering of measurables may restrict the range of possible deliberative interactions.[11]

3.2. **C++ back end.** The BE is in charge of maintaining each BUA's data structures through controlling the parallelized computations executed by the CUDA[12] kernels. Thus, the BE is (1) completely agnostic (in fact, ignorant) of the state of the experiment being run by the FE, with whom it communicates entirely through the RESTful framework; and (2) delegates as much computation as possible to the GPU. While the benefit of the latter to efficiency is clear, one could not overstate the importance of the former: as an agent's BUAs evolve, learn and, potentially, multiply, their network forms an internal representation whose only connection to the agent's environment are the original labeled input streams; thus, an agent could be trained in one environment, while being reasonably expected to continue shaping its network of BUAs to

[11] A rough analogy with classical dynamical systems theory illustrates the nature of this restriction: compare the class of first order real linear dynamical systems in dimension n describable using only upper triangular matrices, with the class of all first order real linear systems in dimension n.

[12] `https://developer.nvidia.com/cuda-toolkit`

accommodate any new situation which provides input streams with the same labeling (and, we assume, the same semantics for that labeling).

Some functions of a BUA—namely, enrichment and pruning—have not yet been delegated to the BE, to enable easy experimentation with different enrichment and pruning algorithms, capitalizing on the versatility of the Python FE.

3.3. **CUDA kernel code.** Our design emphasizes modularity, in particular because the family of different and useful snapshot structures is quite rich (see [4], Section 4). To facilitate modifications to the snapshot structure without affecting any other function of the UMA-SOM architecture, a small number of CUDA kernels was put directly in charge of all snapshot-specific computations (e.g. weight and parameter updating; deriving implications; computing the target state and divergences), and encapsulated in a separate library. Alternative snapshot classes may easily be implemented by editing these kernels and without having to change anything else about the rest of the program package.

3.4. **RESTful communication framework.** Communications between FE and the BUAs in the BE of the UMA-SOM package are realized using the Casablanca[13] RESTful framework. REST, introduced in [1], is an addressing and messaging framework allowing JSON-coded[14] data to be passed between processes through a computer's I/O port, and enabling the distribution of a computational task among an array of event-driven *listeners* on multiple machines, if needed.

In our case, the listeners are the C++/CUDA implementations of the individual BUAs, communicating with the Python FE. The weak typing in Python enables a wide variety of ways to control the BUAs decision and update cycle, as well as the output from each such cycle being returned to FE for use by the experiment or by other BUAs[15].

3.5. **Further Development.** Further improvements to the architecture are possible, especially as theoretical results regarding specific control mechanisms are obtained. We list these development goals in the order in which we intend to pursue them:

BUA Encapsulation: The current version puts excessive computational and memory requirements on the main CPU thread realizing the FE (recall Section 2.1.1). Instead, we envision each BUA 'wrapped' in its own, dedicated "mini-experiment", enabling a distributed computation of the global state update—though at the cost of some memory excesses due to duplication—but with the clear benefit of having *a-priori* bounds on the resources (memory, time) required by each computational thread. This architectural change paves the road to (1) deployment on large clusters, resulting in significant reductions in simulation run times and speedup of our empirical studies; and to (2) effective resource budgeting—a crucial feature for transition to practice, which is currently absent from the architecture.

Saving and Restoring of an Agent: The ability to train an agent in, say, a simplified simulated domain, save its internal state, and then 'release' the agent in some other domain—possibly a more complex one— while maintaining the agent's ability to learn is one of the greater promises of the UMA-SOM architecture. The current RESTful implementation was designed specifically to provide the capability for saving and restoring the BE implementations of all the BUAs in a given experiment. However, the "hooking-up" of these data structures with a given FE user-provided script has not been completed for the lack of a more modular ID registration system. Since such a registration/addressing system will be necessary for the purpose of achieving BUA encapsulation, we consider the save/restore functionality as secondary to the BUA encapsulation task.

[13] `https://github.com/Microsoft/cpprestsdk`

[14] `https://buildbot.tools.ietf.org/pdf/rfc8259.pdf`

[15] In particular, binary queries may be formulated at run time, whose values depend on the internal deliberation of existing BUAs, to be used in the deliberation of any BUA

Migrating Function from FE to BE and Optimizing BUAs: In the current implementation, control over enrichment and pruning function (see stage (5) of the BUA decision cycle in Section 2.1.2) is given over to the front end, to facilitate a broader range of possibilities for experimentation. However, once theoretical results inform the choice of a clear-cut procedure, a BE implementation thereof will provide opportunities for optimizing all computations, as well as for simplifying communication protocols among BUAs and between BUAs and the FE architectural component(s).

4. Some Experiments

4.1. **Available Environments.**

4.1.1. *'Sniffy'.* A simple setting of an agent moving along a discretized interval in search of a single target. The agent's position $\texttt{pos} \in \{0, 1, \ldots, N\}$ on the interval is sensed using a system of 'GPS sensors', $\texttt{x}_i$, $i = 1, \ldots, N$, indicating whether the agent occupies a position strictly to the left of position i. Available actions are moving one step to the left/right, with hard-wired arbitration flipping an even coin between them whenever the agent decides to engage in both. Finally, the motivational signal for either BUA is a monotone-decreasing function φ of the distance to the target, and a sensor `nav` is introduced with $\texttt{nav}|_{t+1} = \texttt{True}$ if and only if $\varphi|_{t+1} > \varphi|_t$.

This UMA-SOM version of the very effective agents (also called 'Sniffy') we had considered in [4] was mainly used for for debugging, as the target implication structures for both BUAs are known, and are fairly simple, e.g.:

$$\texttt{lt}\colon \begin{cases} \texttt{x}_{i-1} < \texttt{x}_i & i = 1, \ldots, N \\ \sharp\texttt{x}_{i-1} < \sharp\texttt{x}_i & i = 1, \ldots, N \\ \sharp\texttt{x}_i < \texttt{x}_{i-1} & i = 1, \ldots, N, \end{cases} \qquad \texttt{lt}^*\colon \begin{cases} \texttt{x}_{i-1} < \texttt{x}_i & i = 1, \ldots, N \\ \sharp\texttt{x}_{i-1} < \sharp\texttt{x}_i & i = 1, \ldots, N \\ \sharp\texttt{x}_i < \texttt{x}_{i+1} & i = 0, \ldots, N-1. \end{cases}$$

Note how $\texttt{lt}^*$ does *not* provide $\sharp\texttt{x}_i < \texttt{x}_i$, due to the possibility of `rt` acting at the same time with $\texttt{lt}^*$. Let us denote these experiments by $\texttt{Sniffy}(N)$, for future reference.

Due to the 'forgetful' nature of the discounted update (recall Equation (2), showing that weights of unobserved states decay with time), having the agent locate the target requires very careful shaping of the motivational signal as a function of the parameter N and of the BUA's learning parameters. This is most evident in simulations where the BUAs infer their own target according to Equation (3) (another option is to supply the fixed target state $\texttt{Targ}^\beta|_t = \{\texttt{nav}\}$ for $\beta \in \{\texttt{rt}, \texttt{rt}^*, \texttt{lt}, \texttt{lt}^*\}$). In this mode (which, admittedly, the intended default for all future BUA implementations), one needs to strike a delicate balance between the size of the environment, the weight decay rate q_α, $\alpha \in \{\texttt{rt}, \texttt{lt}\}$, and the rate at which the signal φ changes as Sniffy approaches its target. Overall, it becomes clear that the reason for this is that practically all of Sniffy's sensorium addresses the environment through a global frame. As a result, the larger the value of N, the larger is the burn-in period (training by randomized actions) required for the computed targets of all BUAs to end up close enough to the real target, that there is a significant probability—once the BUAs have been handed control of the agent—of the computed targets to converge to the real one faster than the agent's position converges to one of the computed targets (once this has happened, the agent is likely to stop moving). This insight is one of the motivations for us having switched to "body frame" sensing models in the 'mice' environment, described next.

4.1.2. *'Mice'.* In a nutshell, the 'mice' simulation environment allows the user to define a rectangular grid world, where every position is given some *attributes*, and may be occupied by *objects*. Agents in this world are embodied as objects called *mice*, each initially endowed with three BUAs—`lt`, `rt`, and `fd`—responsible for the actions of left and right 90° turns, and a forward step along the grid, respectively. The sensing done by a mouse is strictly local, relative to its "body frame", with all initial binary sensors being functions of the content history of a square 'window' centered at the mouse's current position and oriented to match its pose. This is a departure from the construction paradigm we had used earlier for 'Sniffy', but a necessary one if we are to consider more realistic model agents.

We think of this setting as one allowing us to run 'mice' through mazes of varying geometric complexity (by prescribing some positions to serve as obstacles, i.e. positions no mouse could occupy), motivating their BUAs to perform simple tasks. Our focus so far has been on 'find the cheese' tasks: extending the 'Sniffy' setting, multiple 'cheeses' strewn around the environment additively contribute to a "smell landscape"; the

`fd` BUA is then motivated to improve some localized measure of the "current amount of smell", while the `rt` and `lt` BUAs are motivated to minimize the angle between the mouse's pose vector and a discrete gradient of the smell landscape it perceives in its sensory window.

Some degree of arbitration is required to prevent the turning BUAs from interfering with each-other's learning. So far we have implemented a hard-wired deliberation mechanism that (1) flips an even coin to break impasses of the kind $(\texttt{lt} \in \texttt{dec}) \wedge (\texttt{rt} \in \texttt{dec})$ (the mouse wants to turn both right and left), and (2) decides whether to step or to turn in the case when both $\texttt{fd} \in \texttt{dec}$ and $(\texttt{lt} \in \texttt{dec}) \vee (\texttt{rt} \in \texttt{dec})$ (the mouse wants to both step and turn). The latter kind of impasse is resolved by a coin flip as well, but different biases, denoted in Section 4.2.2 below by K, were considered.

While seemingly a toy setting, the 'mice' environment was picked to serve two purposes:

Higher level testing of UMA-SOM implementations: While unit testing assesses the proper functioning of individual procedures, constructing automated tests for an implementation of UMA-SOM as a whole is an impossible task—in part due to the presence of randomization in the decision cycle of every BUA. Therefore, in addition to unit testing, one needs to construct a simulation environment for which specific emergent behaviors of agents are guaranteed to some degree (e.g., on average). The results in Section 4.2.2 indicate that even very simple agents are capable of learning to locate cheeses (by following scent gradients), starting from motor babble, provided that environment has sufficiently simple geometry and trivial topology (e.g., is itself a cubing where the agent traverses the edges). We are currently working on characterizing such 'gradient climbers', to enable the use of mouse agents as a higher order testing/debugging tool for future UMA-SOM implementations.

Controlled experimentation on problems of arbitrary computational complexity: The UMA formalism characterizes BUA planning failures as manifestations of *essential obstacles*—forbidden regions—in their models spaces. In the 'mice' environment, where the 'world' grid is itself a cubing, there is a direct correspondence between physical obstacles (the walls of the maze) and the essential ones. It then follows from results on the limitations of perceptron representation of subset classes in the integer grid [7], that the mere problem of a single 'mouse' agent learning to navigate a maze could incur arbitrary memory and time resource commitments, given a large enough maze and a sufficiently complicated set of obstacles. This makes the 'mice' environment a fruitful—and yet tightly controlled—experimentation rig on which to hone the UMA-SOM architecture.

4.2. **Experimental Results.**

4.2.1. *BUA performance test ('Sniffy').* Due to the specifics of our implementation (e.g. we did not attempt to leverage the possible sparsity of the poc graph), the performance of individual BUAs as a function of sensorium size is independent of the specific experiment being run, provided BUAs of arbitrary sizes could be constructed.

Therefore, $\texttt{Sniffy}(N)$ agents (Section 4.1.1) operating under a random exploration policy could be run for multiple values of N (it is important to stress that the BUAs still perform a complete evaluation of each cycle—it is just that the front end overrides their decisions by random ones), while we collect information about the duration of each decision cycle—for each separate BUA as well as for the architecture as a whole.

By the size of a BUA α at time t, we mean the quantity:

$$\texttt{size}(\alpha)|_t := \max\left(|\mathbf{\Sigma}^{\alpha}|_t|\,, \left|\mathbf{\Sigma}^{\alpha^*}|_t\right|\right)\,. \tag{4}$$

Since the time complexity of a BUAs decision cycle is known to be polynomial in its size, it seems most informative to plot the log-sizes of $\texttt{Sniffy}(N)$ BUAs, for multiple values of N, against the durations of their decision cycles, together with the line of best fit, while comparing these to plots of update cycle durations for the full experiment against experiment sizes (the number of registered measurables).

We have run two batches of experiments, labelled A and B, each consisting of 50 runs of $\texttt{Sniffy}(N_i)$ with enrichment turned on, $i \in \{A, B\}$, with $N_A = 50$ and $N_B = 100$. In each decision cycle of a run of $\texttt{Sniffy}(N_i)$, enrichment may introduce at most one additional query to the active snapshot of every BUA, increasing its size by 2 (the new query and its negation), up to a limit of 2^{N_i+1}. Our choice to enforce a random exploration policy seems more likely to accrue such additions than applying any of the currently available target-seeking policies (which, if successful, limit the BUAs' exposure to the environment). In summary,

with sufficiently long runs we anticipate covering a fairly large interval of sensorium sizes. Figure 3 provides the results produced with runs of length 1000 cycles each on a Microsoft Surface Book laptop with 16GB of RAM and a performance base with an NVIDIA-GTX960 GPU with 2GB of memory (in fact, no more than 0.15GB of GPU memory had been used at any one time).

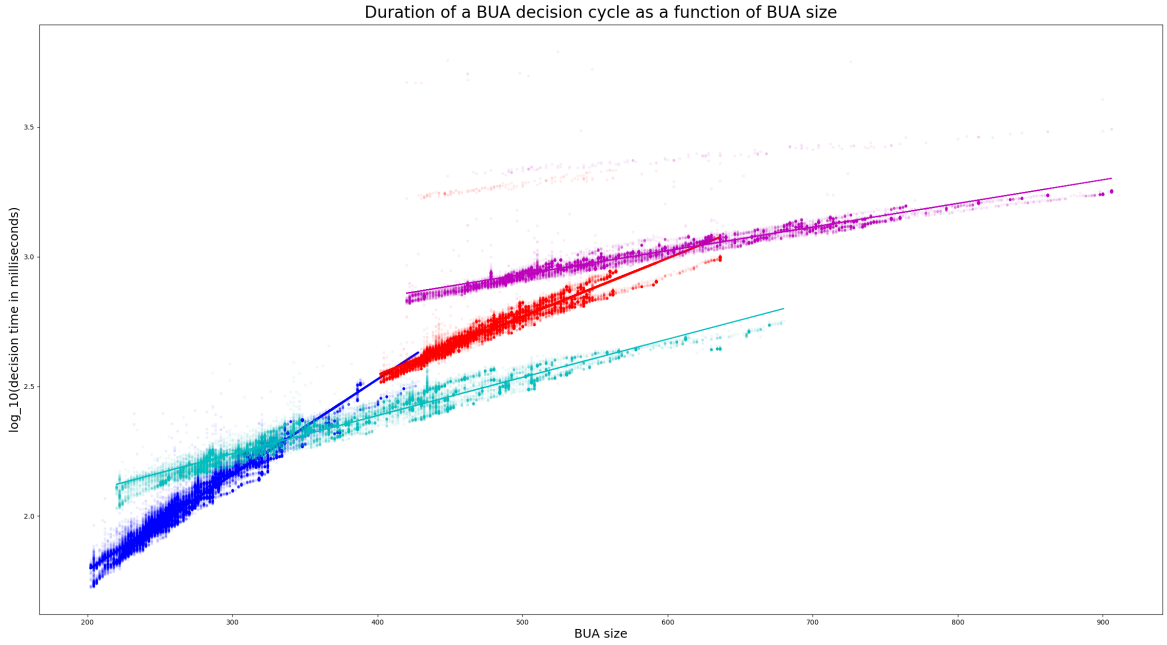

FIGURE 3. Base-10 logarithms of BUA decision cycle durations in milliseconds (blue, red) and experiment update cycle durations (cyan, magenta) plotted, respectively, against BUA sizes as defined in Equation (4) and experiment sizes, with lines of best fit: `Sniffy`(50) (batch A, BUAs in blue, experiment in cyan) and `Sniffy`(100) (batch B, BUAs in red, experiment in magenta), see Section 4.2.1. The transparency of data point markers was set to 5% to make the distribution of data points more visible. Note the apparent decrease in slope when switching from batch A to batch B.

4.2.2. *Learning in 'Mice'.* To assess the sensitivity of 'mice' to the different simulation parameters, we have run the following simulation regimen. For each combination of:

BUA weight discounting parameter: $Q = 1 - 2^{-q}$, $q = \{5, 6, 7, 8, 9\}$;

Stepping *vs.* **Turning bias parameter:** $K = \frac{k}{10}$, $k = \{4, 5, 6, 7, 8\}$, corresponding to $40, 50, \ldots, 80\%$ chance of choosing `fd` over `rt` $\vee$ `lt` in the case of an impasse;

Size of test environment: $D \times D$, with $D \in \{40, 60, 100\}$;

Agent decision-making policy: Four modes of operation were considered for the agent's BUAs:

(1) randomized exploration (BUAs learn, but their decisions are overridden by randomized instructions);
(2) BUAs are assigned fixed target sensations corresponding to high values of their motivational signal, but the signal accumulated in the weight matrices is constant (equals 1);
(3) BUAs are assigned the same target sensations, and the weight matrices integrate the motivational signals;
(4) BUAs infer both their weights and autonomously compute targets from their motivational signals.

we performed 25 simulation runs, each starting with a 'fresh' mouse agent with a window size of 11×11, in two stages:

Training run: The agent is placed in a 20×20 grid, and performs random actions (corrected by the internal arbitration) for 5000 cycles, allowing the BUAs to learn;

Test run: The same agent is then placed in a new test environment, and is run by its BUAs for 5000 additional cycles.

Additional characteristics:

- All runs were initialized with the environment containing a randomized collection of cheeses with an expected number of 0.75 cheeses per window.
- To entice the agents to search for new targets, every time an agent would stay within 3 steps from a cheese (or several cheeses) for at least 50 consecutive cycles, the cheeses in question were removed from the environment.

Figures 4-6 provide a sample of the results, indicating that, while 'mindless' randomized exploration guarantees a steady (linear) decline in the number of cheeses on average, having the BUAs engage in internal deliberation leads, on average, to a faster paced detection and removal of neighboring cheeses, inevitably followed by a reduction in activity as a result of the motivational signal flattening out around the agent. This clearly happens the fastest for the agents inferring their targets directly from their motivational signals. It is also evident, however, that all four policies are roughly equally noisy. Thus, two research questions arise, which must be addressed for some sufficiently broad class of settings generalizing this one:

Question A: What sensory endowment of the BUAs is necessary for guaranteeing swift convergence of the above policies to the policy of climbing the gradient of the motivational signal in the absence of obstacles?

Question B: How to add a curiosity component to the motivational signal, achieving the effect that the autonomous targeting policy would result in random exploration in regions where the raw motivational signal is near-constant?

References

[1] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000. `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`.

[2] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962. `https://dl.acm.org/citation.cfm?doid=367766.368168`.

[3] D.P. Guralnik and D.E. Koditschek. Toward a memory model for autonomous topological mapping and navigation: The case of binary sensors and discrete actions. In *Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on*, pages 936–945, 2012. `http://ieeexplore.ieee.org/abstract/document/6483319/`.

[4] D.P. Guralnik and D.E. Koditschek. Universal memory architectures for autonomous machines. *preprint*, 2016. `https://repository.upenn.edu/ese_papers/737/`.

[5] Marvin Minsky. K-lines: A theory of memory. *Cognitive science*, 4(2):117–133, 1980. `http://onlinelibrary.wiley.com/doi/10.1207/s15516709cog0402_1/full`.

[6] Marvin Minsky. *The Society of Mind*. Simon & Schuster, 1988.

[7] Marvin Minsky, Seymour A Papert, and Léon Bottou. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

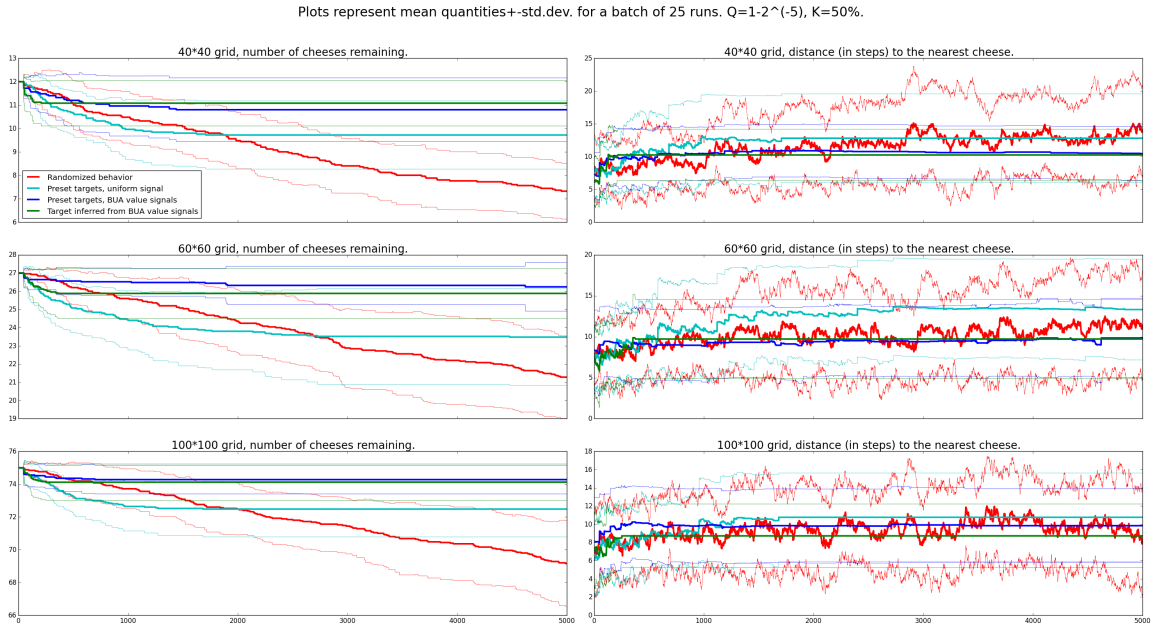

FIGURE 4.

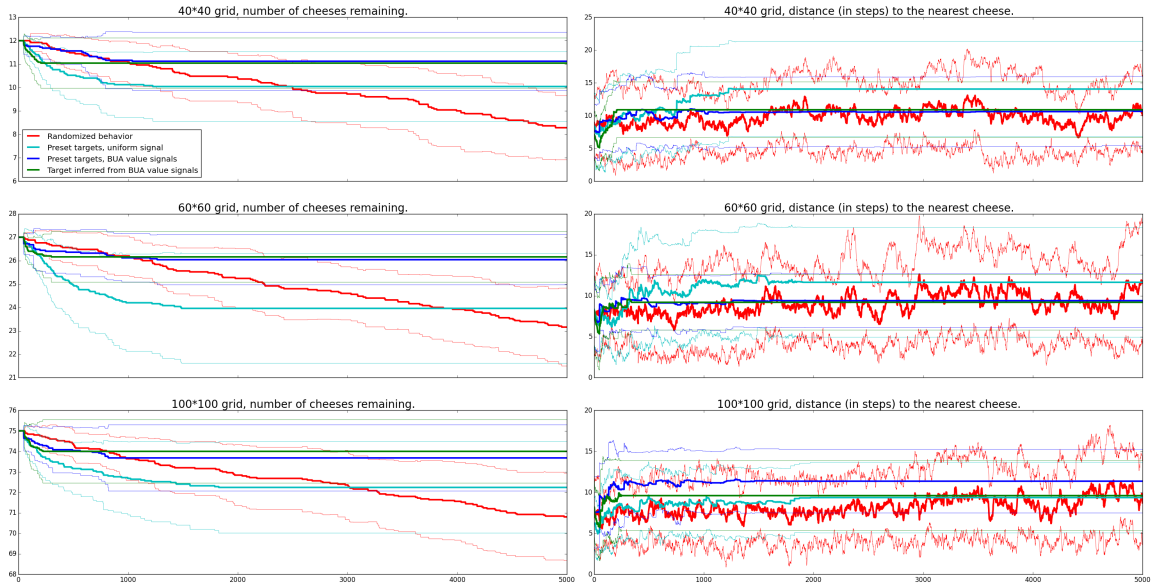

FIGURE 5.

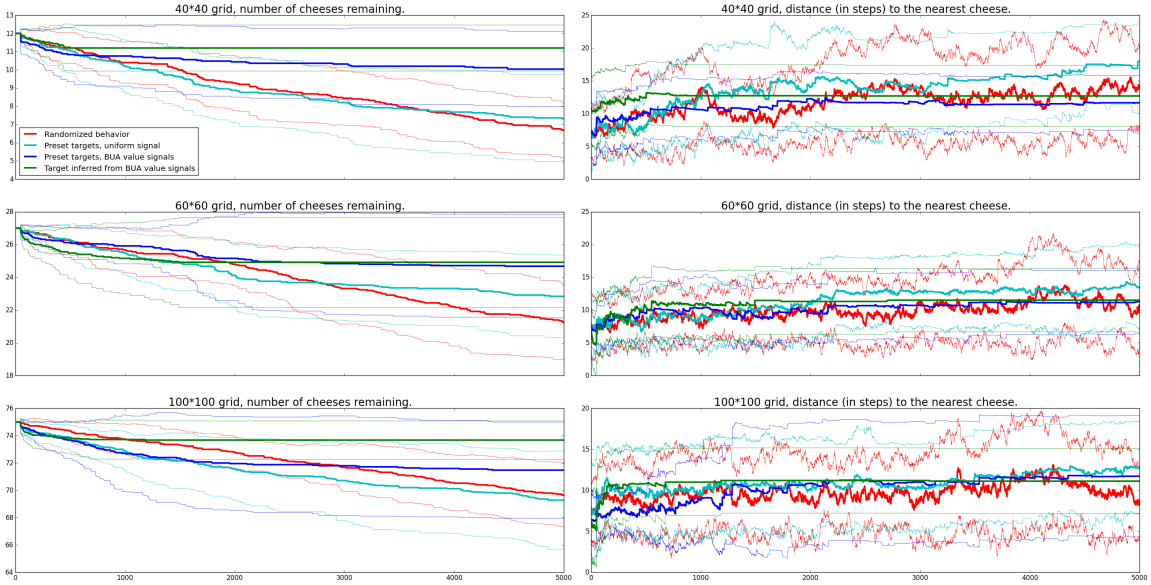
Plots represent mean quantities+-std.dev. for a batch of 25 runs. Q=1-2^(-9), K=50%.
40*40 grid, number of cheeses remaining.
40*40 grid, distance (in steps) to the nearest cheese.
Randomized behavior
Preset targets, uniform signal
Preset targets, BUA value signals
Target inferred from BUA value signals
60*60 grid, number of cheeses remaining.
60*60 grid, distance (in steps) to the nearest cheese.
100*100 grid, number of cheeses remaining.
100*100 grid, distance (in steps) to the nearest cheese.

FIGURE 6.