Dynamic State Space Partitioning for Adaptive Simulation Algorithms

Tobias Helms
Institute of Computer Science
University of Rostock
18059 Rostock, Germany
tobias.helms@uni-rostock.de

Steffen Mentel
Institute of Computer Science
University of Rostock
18059 Rostock, Germany
steffen.mentel@uni-rostock.de

Adelinde M. Uhrmacher
Institute of Computer Science
University of Rostock
18059 Rostock, Germany
adelinde.uhrmacher@uni-rostock.de

ABSTRACT

Adaptive simulation algorithms can automatically change their configuration during runtime to adapt to changing computational demands of a simulation, e.g., triggered by a changing number of model entities or the execution of a rare event. These algorithms can improve the performance of simulations. They can also reduce the configuration effort of the user. By using such algorithms with machine learning techniques, the advantages come with a cost, i.e., the algorithm needs time to learn good adaptation policies and it must be equipped with the ability to observe its environment. An important challenge is to partition the observations to suitable macro states to improve the effectiveness and efficiency of the learning algorithm. Typically, aggregation algorithms, e.g., the adaptive vector quantization algorithm (AVQ), that dynamically partition the state space during runtime are preferred here. In this paper, we integrate the AVQ into an adaptive simulation algorithm.

Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Adaptive Simulation*; I.2.6 [**Artificial Intelligence**]: Learning—*Reinforcement Learning*

Keywords

Adaptive Algorithms, Reinforcement Learning, Component-based Simulation Software, Dynamic State Space Representations

1. INTRODUCTION

Adaptive simulation algorithms change their configuration during runtime automatically to improve the overall performance of a simulation [3]. Adaptations can be necessary due to changing computational demands during a simulation run, e.g., caused by a changing number of entities or the execution of a rare event. In general, it is challenging for a user to select a suitable algorithm and configuration for a specific experiment, let alone changing the algorithms during runtime. Even developer of complex algorithms themselves are typically not able to evaluate the performance of their algorithms for a specific problem properly [5]. Adaptive algorithms are an approach to deal with these challenges.

Machine learning techniques can be used to learn when to use which configuration automatically. Adaptive algorithms that use such techniques can be reused for different models, modeling languages and simulation experiments [3]. A suitable machine learning technique is reinforcement learning [11]. In reinforcement learning, the following procedure is repeated until a termination criterion is fulfilled, e.g., a simulation run terminates. An agent observes the environment and performs an action. Then, it receives a reward from the environment and it updates its knowledge base. Typically, the agent starts with little knowledge about the environment and it must find a suitable trade-off between the exploration and the exploitation of knowledge. Based on reinforcement learning, we developed a generic adaptive simulation algorithm [3].

However, the advantages of the learning algorithms come with a cost. For example, they must automatically determine how the features of the model and the environment influence the performance. In reinforcement learning, aggregation algorithms are used to deal with this problem, e.g., [9, 10, 6]. These algorithms partition the state space into disjunct macro states dynamically. A macro state represents a region of the state space with a homogeneous performance behavior. All states within the same macro state are treated equally, i.e., knowledge learned for one state is reused for all other states of the same macro state. The performance of these aggregation algorithms strongly depends on the given problem and the used configuration — a proper usage of these algorithms is a challenge in itself. In this paper, we discuss several aggregation algorithms referring to their applicability for adaptive simulation algorithms and we adjust the adaptive vector quantization algorithm [6] so that it can be used for the adaptive simulation algorithm developed in [3].

2. ADAPTIVE SIMULATOR

The adaptive simulation algorithm called "adaptive simulator" developed in [3] uses reinforcement learning to learn when to use which simulation algorithm and configuration. Referring to reinforcement learning, the adaptive simulator represents the agent, all available simulation algorithms and configurations represent the available actions, and all observable information like the model state represent the environment. The adaptive simulator is organized as a wrapper for other simulation algorithms, i.e., it uses other algorithms to compute the actual simulation and it exchanges the current simulation algorithm during an adaptation. The available simulation algorithms and configurations are listed in a finite

VALUETOOLS 2015, December 14-16, Berlin, Germany

DOI 10.4108/eai.14-12-2015.2262710

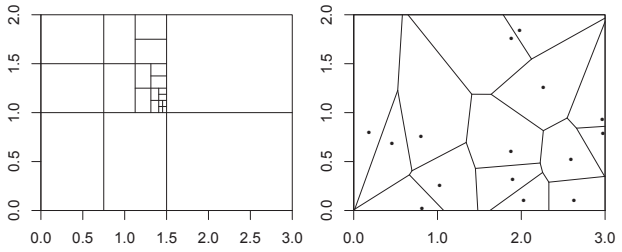

Figure 1: Left: Exemplary partitioning created by the decision boundary partitioning algorithm. Right: Exemplary partitioning created by vector quantization represented by a Voronoi diagram. Each point represents one of the current codewords.

set $A = \{a_1, a_2, \ldots, a_n\}$. During a simulation, simulation events are executed and a data vector ("base state") $\sigma \in \Sigma$ of data is observed and appended to a base state trajectory $\tau \in \Sigma^*$ after each event execution (Alg. 1, l. 9-11). When the adaptation condition is fulfilled (l.12), an adaptation is triggered. The adaptation condition function uses a Bayesian changepoint detection algorithm to determine adaptation points [4]. When an adaptation shall be executed, the adaptive simulator firstly computes the current reward $r \in \mathbb{R}$ (l. 13), e.g., the event throughput. The knowledge base is updated (l. 23) by using Q-Learning that learns so called q-values $q \in \mathbb{R}$ for each state-action pair (s, a), representing the utility to select a after observing s [14]. Afterward, the current state $s \in S$ is computed by using the base state trajectory $\tau \in \Sigma^*$. The state is then mapped to a macro state that represents a region in the state space. So far, we only use static grids to partition the state space. This approach is simple but it has obvious disadvantages, i.e., finding a suitable scale for the grid is difficult and problem-dependent. Furthermore, different granularities are usually needed in different areas of the state space. Finally, the new simulation algorithm is selected by using a selection policy (l. 29) and the current simulation algorithm is exchanged with the new one (l. 31). For the selection policy, we use ϵ-decreasing [12] that is a simple but robust and efficient policy, see [2, 3].

3. AGGREGATED STATE SPACES

Due to high-dimensional and real-valued state spaces, it is usually not feasible to learn a suitable selection policy for each state individually. Aggregation algorithms, e.g., [9, 10, 6, 13, 1], dynamically partition the state space of a reinforcement learning problem into disjunct macro states do deal with this problem. Typically, these algorithms start with a coarse-grained partitioning of the state space and refine the state space based on various conditions.

For example, the **parti-game algorithm** [9] assumes that *a*) the goal is to reach a specific region of the state space, *b*) that there exists a continuous path from the start position to the goal region, *c*) that state transitions are deterministic, and *d*) that the agent can move deliberately through the state space. Due to these requirements, this algorithm cannot be applied to the adaptive simulator. First, the goal of the adaptive simulator is not to reach a specific region of the state space, but to maximize the received rewards. Second, the adaptive simulator cannot move deliberately through the state space — it cannot influence deliberately model variables that are possibly used as dimensions for the state space.

An extended and less restricted version of the parti-game algorithm is the **decision boundary partitioning algorithm** [10]. This algorithm splits two adjacent macro states ms_i and ms_j at the midpoint of their longest dimension if the following three conditions based on the current knowledge hold. First, the best action in both macro states differ. Second, the difference of the utilities of the best action of ms_i and the best action of ms_j is in ms_i or ms_j higher than $\Delta_{min} \in \mathbb{R}$. Third, all actions of both macro states have been visited at least v_{min} times. The conditions guarantee that only reasonable splits of macro states are executed. Figure 1 (left) illustrates an exemplary state space partitioning that could have been built with the decision boundary partitioning algorithm. This algorithm can be used for the adaptive simulator, but there are various challenges. For example, if many observed states occur inside the same macro state but not inside its neighbors, this algorithm will not split this macro state although it could be useful. In the worst case, no splits are executed at all because the initial macro states have been set poorly.

Another group of aggregation algorithms uses the idea of the nearest neighbor vector quantization to create macro states, e.g., [6, 13]. These algorithms maintain a codebook $CB \subseteq S$ containing specific states that are called codewords. A nearest vector quantizer is used to map a state $s \in S$ onto the nearest codeword $c \in CB$ available in the current codebook. Basically, this mapping creates a partitioning of the state space into disjoint regions, i.e., the macro states (see Figure 1). The idea of these algorithms is to frequently change the codebook, so that all states mapped to a macro state represent a similar q-value behavior. A promising algorithm of this group to be applied to the adaptive simulator is the **adaptive vector quantization algorithm (AVQ)**, because a) it does not depend on the definition of a goal state, b) it allows complex shapes of macro states, and c) no initial partitioning for each dimension must be defined. Further, compared to other aggregation algorithms that use the nearest neighbor vector quantization, the configuration effort of this algorithm is low. To decide whether new codewords shall be added to the codebook, this algorithm uses a concept based on the accumulated reward ($accReward$), that "with respect to a particular action is the sum of the total rewards received by continuously taking the same action within a particular cell" [6]. Consequently, this algorithm directly influences the action selection. This approach has to be replaced when applying this algorithm to the adaptive simulator, because it is possible that a poor performing action is reused frequently.

4. AVQ AND ADAPTIVE SIMULATOR

Algorithm 1 illustrates the integration of the AVQ algorithm in the adaptive simulator. As motivated above, we replaced the concept of the accumulated reward. Instead, we use a condition inspired by the decision boundary partitioning algorithm. A state s, that is mapped to a codeword c, is added to the codebook if the absolute difference of the current reward and the last reward achieved by the same action for any other state s' mapped to c is higher than a threshold $\alpha \in \mathbb{R}$ (see l. 16). Generally speaking, a state is added to the codebook if the rewards of an action differ significantly within its macro state. Further, we add a limit $m \in \mathbb{N}$ for the number of macro states. The reward of the adaptive simulator is calculated by computing the logarithm of base 2 of the event throughput. We set $\alpha = log_2(1.5) \approx 0.585$ by default, i.e., a throughput difference of at least 50% must occur so that a codeword is added to the codebook. We reused the merging routine of the AVQ (l. 35-41). Thus, at the end of each simulation run, i.e., after finishing the adaptation loop, the merging routine is executed. We set $\rho = \alpha^2 \approx 0.342$ by default, so that this parameter is linked to the condition to add codewords to the codebook.

We demonstrate the effectiveness and efficiency of our approach with a benchmark using a two-dimensional state space $(x, y) \in$

Algorithm 1 Pseudo-code for the adaptive simulator [3] extended by the changed AVQ.
Q: q-value matrix indexed by state $s \in S$ and action $a \in A$.
N: matrix of counters for visited (s, a) tuples.
$s, s' \in S$: previous and current state.
$a \in A$: action. $r \in \mathbb{R}$: reward.
$R : \Sigma^* \to \mathbb{R}$ reward function.
$\sigma \in \Sigma$: current base state.
$\tau \in \Sigma^*$: current base state trajectory (seq. of base states).
$p : \Sigma^* \to S$: base state trajectory transformation function.
$f : S \times \mathbb{R}^{|S|\times|A|} \times \mathbb{N}^{|S|\times|A|} \to A$: action selection policy.
$L : CB \times A \to \mathbb{R}$: reward matrix containing the last rewards indexed by codewords and actions

```
c := nearest_neighbor(s_0, CB)
s := s_0
a := a_0
initialize(a)
repeat { //Adaptation loop
  N[c,a] := N[c,a] + 1
  τ := []
  repeat { //Simulation loop
    simulate_next_event(a)
    σ := observe(a)
    τ := τ + σ
  } until adaptation_condition(τ, f, R)
  r := R(τ) //Calculate reward

  //Check whether to add s to the codebook
  if (|CB| < m && |L[c,a] − r| > α) {
    CB := CB ∪ {s}
    c := s
  }
  L[c,a] := r

  //Simplified Q–learning:
  Q[c,a] := Q[c,a] + 1/(N[c,a]) · (r − Q[c,a])
  //Aggregate base states to a state
  s := p(τ)
  //Get codeword for computed state
  c := nearest_neighbor(s, CB)
  //Select next action based on c, Q, N:
  a_next := f(c, Q, N)
  //Adapt internal SA:
  a := adapt(a, a_next, σ)
} until is_terminal(s)

//Merging process
for (c ∈ CB) {
  c' := nearest_neighbor(c, CB)
  if ((Σ_{a∈A}(Q[c,a] − Q[c',a])^2) ÷ |A| < ρ) {
    CB := (CB\{c, c'}) ∪ {(c + c')/2}
    update_B(Q, N, c, c')
  }
}
```

$[0, 2\pi] \times [-1, 1]$. Here, the state space is partitioned by a sine curve and a cosine curve. The action set of the agent contains three actions a_1, a_2, and a_3. The reward calculation works as follows. If the state observed by the agent is above the sine and cosine curve, it receives reward 1 for choosing action a_0, and reward 0 for the other actions. If the state observed by the agent is between the sine and cosine curve, it receives reward 1 for choosing action a_1, and reward 0 for the other actions. In the last case, i.e., if the state observed by the agent is below the sine and cosine curve, it receives reward 1 for choosing action a_2, and reward 0 for the other actions. The actions chosen by the agent do not influence the next state of the environment — the next state is chosen randomly. Finally, 100 trials are executed with 1000 steps per trial, i.e., the merging process is executed every 1000 steps. Although simple, this setting reflects important aspects of the adaptive simulator scenario, e.g., the actions of the agent do not influence the observed environment states. Figure 2 illustrates the final partitioning of the state space using $m \in \{10, 100\}$. The success rates (the rate of correct decisions) are shown in Figure 3 (left). As expected, the more macro states are allowed, i.e., the higher m is chosen, the more accurate gets the state space partitioning and the higher is the success rate. However, the success rate converges to a value ≈ 0.87, i.e., at some degree it is not worth to increase the number of macro states, because most of the occurring states are already covered. Figure 3 (right) also shows the success rate of a static rectangular grid with different macro state sizes. Here, the smaller the length and width of a macro state, the worse the success rate, because the more redundant knowledge is learned.

To demonstrate that the results do not depend on the granularity of state space dimensions, we repeat the same benchmark experiment with a scaled down state space with scaled down sine and cosine curves so that $x \in [0.52, 0.54]$ and $y \in [0.39, 0.41]$. The results of this scaled down benchmark are also shown in Figure 3 (crosses). Whereas the results using the changed AVQ are almost equal, the results using a static grid state space differ significantly, i.e., a granularity that is effective for one problem is usually unsuitable for another.

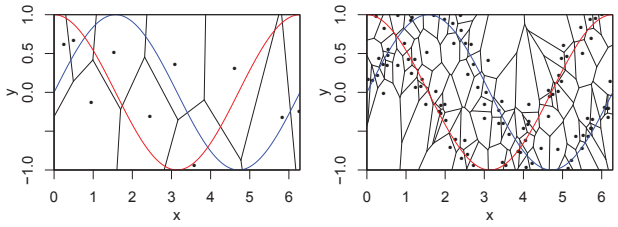

Figure 2: State space partitionings built by using the changed AVQ with $m = 10$ (left) and $m = 100$ (right).

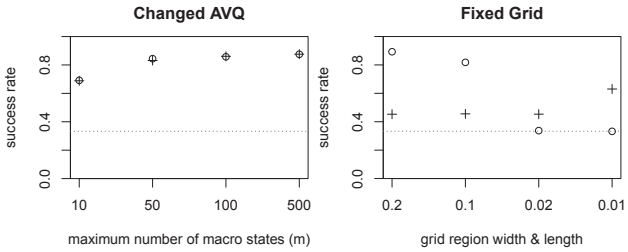

Figure 3: Success rates for the benchmark using the changed AVQ (left) and a fixed grid (right), once for the size $x \in [0, 2 \cdot \pi], y \in [-1, 1]$ (circles), once for the scaled down size $x \in [0.52, 0.54], y \in [0.39, 0.41]$ (crosses). The dotted lines show the success rate of a random choice, i.e., $\frac{1}{3}$.

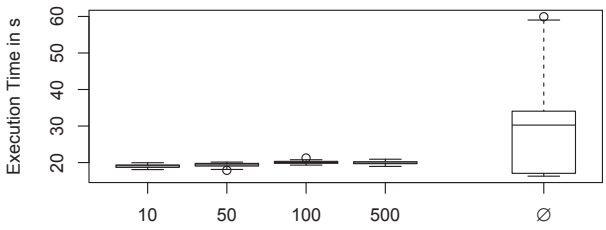

Figure 4: Execution times of the Wnt/β-catenin pathway model with the adaptive simulator using the changed AVQ with $m \in \{10, 50, 100, 500\}$ compared to the execution times of the available 24 non-adaptive simulator configurations.

5. ML-RULES SIMULATION

Besides the benchmark, we executed a Wnt/β-catenin pathway model [8] implemented in the modeling language ML-Rules [7] to evaluate the adaptive simulator using the changed AVQ. ML-Rules is used to model biochemical reaction networks. For ML-Rules, various simulation algorithms and components exist, e.g., to manage the species and reactions sets. Here, we selected 24 configurations to execute ML-Rules simulations available for the adaptive simulator, i.e., $|A| = 24$. Besides different state space partitioning configurations, we used the default configuration of the adaptive simulator. For the changed AVQ, we used $m \in \{10, 50, 100, 500\}$. Each simulation run of the Wnt/β-catenin pathway model was executed for 100,000 simulation events. We sequentially executed 100 replications of this simulation. The knowledge base and the state space partitioning of the adaptive simulator were reused over the replications. Consequently, referring to the merging process of the state space partitioning, 100 merging processes were executed. After finishing 100 replications, the average runtime to execute one simulation run was calculated. We repeated this experiment 50 times to get a reliable distribution of the average execution times. Figure 4 illustrates these execution time distributions of the adaptive simulator with $m \in \{10, 50, 100, 500\}$. Additionally, the average execution time distribution of the 24 non-adaptive simulator configurations are shown. The adaptive simulator achieves almost the performance of the best non-adaptive simulator configuration with all values of m. For the Wnt/β-catenin pathway model, a small number of macro states is apparently sufficient to achieve good results. However, the results show that the performance of the adaptive simulator only worsen slightly with a higher number of macro states owing to a higher exploration effort. This emphasizes the robustness of dynamic state space partitioning algorithms. Altogether, compared to a random choice of a configuration, the adaptive simulator is significantly more efficient.

6. CONCLUSION

Adaptive simulation algorithms like the adaptive simulator [3] change their configuration automatically during runtime, 1) to improve the overall performance of the simulation, and 2) to relieve the user to configure the simulation algorithm. Using learning techniques for these algorithms make them reusable, however, these techniques come with own challenges that must be solved. In this paper, we dealt with the problem of dynamic state space partitionings [10, 6]. We integrate the adaptive vector quantization algorithm [6] into the adaptive simulator. Our short evaluation has illustrated that applying a dynamic state space partitioning makes the adaptive simulator more efficient and reliable and further reduces its configuration effort.

7. ACKNOWLEDGMENTS

This research was supported by the German research foundation (DFG), via the research grant ESCeMMo (UH-66/14).

8. REFERENCES

[1] A. Bonarini, A. Lazaric, and M. Restelli. Reinforcement Learning in Complex Environments Through Multiple Adaptive Partitions. In *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, volume 4733, pages 531–542. 2007.

[2] R. Ewald, J. Himmelspach, and A. M. Uhrmacher. An Algorithm Selection Approach for Simulation Systems. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS'08)*, pages 91–98, 2008.

[3] T. Helms, R. Ewald, S. Rybacki, and A. M. Uhrmacher. Automatic Runtime Adaptation for Component-based Simulation Algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 2015. to appear.

[4] T. Helms, O. Reinhardt, and A. M. Uhrmacher. Bayesian Changepoint Detection for Generic Adaptive Simulation Algorithms. In *Proceedings of the 48th Annual Simulation Symposium (ANSS'15)*, pages 62–69, 2015.

[5] H. H. Hoos. Programming by Optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[6] I. S. Lee and H. Y. Lau. Adaptive state space partitioning for reinforcement learning. *Engineering Applications of Artificial Intelligence*, 17(6):577–588, 2004.

[7] C. Maus, S. Rybacki, and A. M. Uhrmacher. Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology*, 5(166), 2011.

[8] O. Mazemondet, R. Hubner, J. Frahm, D. Koczan, B. Bader, D. Weiss, A. M. Uhrmacher, M. J. Frech, A. Rolfs, and J. Luo. Quantitative and kinetic profile of Wnt/β-catenin signaling components during human neural progenitor cell differentiation. *Cellular & Molecular Biology Letters*, 16(4):515–538, 2011.

[9] A. W. Moore and C. G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233, 1995.

[10] S. I. Reynolds. Decision boundary partitioning: Variable resolution model- free reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning (ICML'00)*, pages 783–790, 2000.

[11] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[12] J. Vermorel and M. Mohri. Multi-armed Bandit Algorithms and Empirical Evaluation. In *Proceedings of the 16th European conference on Machine Learning (ECML'05)*, pages 437–448, 2005.

[13] D. Vieira, P. Adeodato, and P. Goncalves. A Temporal Difference GNG-Based Algorithm That Can Learn to Control in Reinforcement Learning Environments. In *Proceedings of the 12th International Conference on Machine Learning and Applications (ICMLA)*, pages 329–332, 2013.

[14] C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, England, 1989.