Resilient Computing with Reinforcement Learning on a Dynamical System: Case Study in Sorting

Aleksandra Faust[1] James B. Aimone[2] Conrad D. James[2] Lydia Tapia[3]

arXiv:1809.09261v1 [cs.AI] 25 Sep 2018

Abstract— **Robots and autonomous agents often complete goal-based tasks with limited resources, relying on imperfect models and sensor measurements. In particular, reinforcement learning (RL) and feedback control can be used to help a robot achieve a goal. Taking advantage of this body of work, this paper formulates general computation as a feedback-control problem, which allows the agent to autonomously overcome some limitations of standard procedural language programming: resilience to errors and early program termination. Our formulation considers computation to be trajectory generation in the program's variable space. The computing then becomes a sequential decision making problem, solved with reinforcement learning (RL), and analyzed with Lyapunov stability theory to assess the agent's resilience and progression to the goal. We do this through a case study on a quintessential computer science problem, array sorting. Evaluations show that our RL sorting agent makes steady progress to an asymptotically stable goal, is resilient to faulty components, and performs less array manipulations than traditional Quicksort and Bubble sort.**

I. Introduction

Modern software controls transportation systems, stock markets, manufacturing plants, and other high-consequence systems. The software often runs in operating environments that are vastly different, and changing, from their original scopes [1]. One cause of changes are soft errors, random and temporary errors that affect all aspects of computing, such as memory, registers, and calculations [11]. Undetected and unmanaged, their cumulative effect can be severe. For example, particle and electromagnetic radiation corrupt computing in space, at high-altitude, and around nuclear reactors, causing silent software failures. Similarly, current microchip design pushes the physical limits, causing memory faults and soft computation errors [7]. Traditional radiation hardening and fault-tolerant computing consist of physical system redundancy and material layering. Both increase the complexity and cost. Resilient algorithms, algorithms that adapt and work around soft errors, are gaining popularity [1].

Contemporary applications require algorithms to work with limited resources. For instance, an algorithm might be not be allowed to self-terminate. Rather it might be interrupted to yield to a higher priority problem, and must produce a partial answer no worse than the quality of the input data. Thus, an early termination and steady progress toward the solution become requirements.

Yet, traditional computing works under two fundamental assumptions: correctness and program-initiated termination. Theoretical computer science considers an algorithm *totally correct* if, for a correct input, the algorithm terminates and returns a correct calculation [16]. The total correctness influences traditional software engineering by developing best practices such as code reviews [12], design patterns [4], and correctness proofs [12], intended to guard against faulty specification and developer-introduced bugs. The total correctness does not pose any restriction on how computation evolves over time, the result is correct as long as it is given the correct input. That has two consequences. First, the total correctness offers no adaptation and resiliency to errors in the input or the program's dependencies. Second, there are no guarantees on what the algorithm's outcome might be if the program is terminated early.

In contrast, feedback control and sequential decision making in robotics often require error adaptation, and termination when a sufficient accuracy is achieved. Robots routinely rely on measurements that contain errors, yet still aim at providing *resilient* decision making. Next, in many robot control tasks, it is important how a robot accomplishes a task, not only that it does so. For example, among all the trajectories that take the robot from its current position to the goal, we might want to choose the shortest, safest, or most fuel-efficient path. *Stable* decision making finds trajectories that steadily progress towards the goal with respect to some metric.

This paper applies the *resiliency* and *stability* ideas from controls to computing aiming to overcome correctness and termination limitations. Given the similarities in the operating uncertainties of software and robots, we consider a computation to be a trajectory generation in the program's variable space [16]. To that end, we show that a computer program is a discrete time dynamical system over the vector space defined by its variables, and is controlled with vector transformation matrices with an equilibrium in the correct output. Adding a reward, we arrive at Markov Decision Process formulation, which we solve with reinforcement learning to learn a control Lyapunov function. Lyapunov stability theory analyzes the quality of the resulting controlled system, gives theoretical guarantees for steady progression toward the goal, and

[1]Google Brain, Mountain View, CA, USA, faust@google.com
[2]Sandia National Labs, Albuquerque, NM, USA.
[3]Department of Computer Science, University of New Mexico, Albuquerque, NM, USA.

probability of success in the presence of soft errors. As an example, in order to show how control and decision-making tools can shape algorithm design, we focus on a quintessential computer science problem: array sorting. We model the soft errors as an error-prone comparison [1], [2]. The resulting agent has the desired resiliency and stability properties.

This paper contributes a description of a sequential state-based computational process as a Markov Decision Process where learning can be applied to produce a stable and resilient solution. This is achieved through a novel application of reinforcement learning and Lyapunov stability theory to develop a resilient sorting algorithm in environments with a very high probability of soft errors (up to 50%). Specific contributions of this work are 1) a controlled dynamical system formulation for computing in Section IV, 2) a RL sorting agent in Section IV-B, and 3) stability and resiliency analysis of the proposed algorithm (Section IV-C). The methods presented in this paper are general, and it is expected that they can be generalized to other iterative, resilient computing algorithms.

II. Related Work

Sorting has been studied as a quintessential computer science problem in resilient computing. Specifically, in the context of soft-memory faults due to hardware imperfections [11] and ranking applications due to measuring imprecisions [2]. To sort correctly, the first model requires registers that are never corrupted and the known maximum number of memory corruptions [11]. In contrast, our algorithm requires neither of these. We assume that, at any point, memory corruption is possible with some probability. We model the memory corruption as a p-faulty comparison between two elements, which returns an inaccurate comparison with probability p. Our algorithm does not need to know the corruption probability, and the error rate can change over time, up to 50%. The second error model, ranking, relies on performing multiple rounds on imprecise comparisons. Similar to our method, [2] trades time for accuracy and requires $O(n^2)$ comparisons, while delivering accuracy in probability. RL sort, in addition, prioritizes element selection.

Our solution differs from the previous resilient methods in the applications and tools we use. First, our method makes little to no assumptions about the source and frequency of errors. Faulty comparisons that we choose to work with might be symptomatic of many different causes, such as I/O, transmission, or computation errors. Second, our method uses reinforcement learning to learn the near-optimal element selection policy, which is then used to repeatedly choose element to move and its destination.

Reinforcement programming [20] uses Q-learning to generate a sorting program in absence of errors, using more traditional programming constructs such as counters, *if* statements, *while* loops, etc., as MDP actions. The output is a program ready for execution, tailored for a specific array. Our implementation, in contrast, considers element insertion as the only possible action. Instead of producing an executable program, our agent directly sorts a given array. In other agent-based work, Kinnear uses genetic programming [14], which produces Bubble sort as its most-fit solution.

Error measurement of the intermediate results of traditional sorting algorithms reveals a cyclical structure of errors with respect to several metrics [13]. The error oscillations during program execution mean that the results of traditional sorting algorithms, if interrupted before completion, can be worse than the starting state. Our RL sort makes steady progress, and, if interrupted, is guaranteed to return a more sorted array than the one it started with.

III. Background

Trajectory generation over time can be described as a discrete time, controlled, nonlinear dynamical system at time step n,

$$D: \quad \boldsymbol{x}_{n+1} = \boldsymbol{f}(\boldsymbol{x}_n, \boldsymbol{u}_n), \tag{1}$$

for a nonlinear function $\boldsymbol{f} : X \times U \to X$, where the system state $\boldsymbol{x} \in X$, and the input, or action, $\boldsymbol{u} \in U$, influences the system, and changes its current state. Here, we consider fully observable systems, i.e., $\boldsymbol{y} = \boldsymbol{x}$.

A deterministic *Markov decision process* (MDP), a tuple (X, U, D, R) with states $X \subset \mathbb{R}^{d_s}$ and action $U \subset \mathbb{R}^{d_a}$, that assigns immediate scalar rewards $R : X \to \mathbb{R}$ to states in X, formulates a task for the system (1) [6]. A solution to a MDP is a control policy $\pi : X \to U$ that maximizes the cumulative discounted reward over an agent's lifetime (state-value function), $V(\boldsymbol{x}(0)) = \sum_{i=0}^{\infty} \gamma^i R(\boldsymbol{x}(i))$, where $0 \leq \gamma \leq 1$ is a discount. System D in (1) is a state transition function.

RL solves MDP through interactions with the system and is appropriate when state transition function D or reward R are not explicitly known [6]. *Approximate value iteration* (AVI) [8] finds a near-optimal state-value function, $V : X \to \mathbb{R}$ approximated with a feature map

$$\hat{V}(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{F}(\boldsymbol{x}) \tag{2}$$

AVI works in two phases, *learning* and *trajectory generation*. The learning phase takes a user-provided feature vector $\boldsymbol{F}(\boldsymbol{x})$ and learns weights $\boldsymbol{\theta}$ in expectation-maximization (EM) manner. After the learning, AVI enters the trajectory generation phase, with an initial state, the feature vector, and the learned parametrization. It creates trajectories using a greedy closed loop control policy with respect to the state-value approximation,

$$\pi^{\hat{V}}(\boldsymbol{x}) = \underset{\boldsymbol{u} \in A}{\operatorname{argmax}}\, \hat{V}(\boldsymbol{x}'), \tag{3}$$

where state $\boldsymbol{x}'$ is the result of applying action $\boldsymbol{u}$ to state $\boldsymbol{x}$, $\boldsymbol{x}' = D(\boldsymbol{x}, \boldsymbol{u})$. AVI was used in a wide class of problems from control of unmanned aerial vehicles (UAVs) [5], to UAVs with a suspended load [9] and electrical power control systems [8], among others.

Lyapunov stability theory gives us tools to assess the stability of an equilibrium. An equilibrium is *globally asymptotically stable in sense of the Lyapunov* if outcomes of any two initial conditions converge to each other over time [3]. The Lyapunov direct method gives sufficient conditions for stability of the origin [3]. The method requires construction of a positive definite scalar function of state $W : X \to \mathbb{R}$ that monotonically decreases along a trajectory and reaches zero at equilibrium. This function can be loosely interpreted as the system's energy, positive and deceasing over time until it is depleted and the system stops. Task completion of a RL-planned motion can be assessed with Lyapunov stability theory, for example, to choose between predetermined control laws in order to guarantee task completion [17], or to construct a state-value function such that it is a control Lyapunov function [9] [10]. We use the latter method here.

IV. Methods

We first pose general computing as a controlled dynamical system in Section IV. Section IV-B focuses on a sorting problem, and develops a RL agent, which is analyzed in Section IV-C.

A. Computing as a Dynamical System

We consider deterministic programs, where a program's outcome is determined by two factors: the initial state of its variables, and the sequence of steps (*algorithm*) that solve the problem. The control-flow constructs, if-then-else, and loops, are controller switches. Instructions are performed at discrete time steps per internal clock tick. The state transitions are determined by executing instructions in the instruction register. A program seen this way is a control policy for a dynamical system determined by the change in the state of variables over time until the computation stops. Computation is a trajectory in the variable space starting with an initial state of variables (initial vector) and terminating at the goal state (goal vector).

At runtime the program's in-scope variables and the location of the instruction counter uniquely determine the program's state (*configuration*) [16]. Regardless of how a program got into a certain configuration, the computation unfolds the same from that point. Thus, a program's state space is the space of all variable values, and satisfies the Markovian property. Without loss of generality, we assume all variables to be real numbers. Thus, a state space for a program with d_s variables is $X = \mathbb{R}^{d_s}$, a d_s-dimensional vector. Operations and programming constructs that change the variables, such as *assignment*, *arithmetic operations*, and *changing instructions*, are the action space. Proposition 4.1 shows that in such a setup a program is a nonlinear dynamical system because the states are vectors and the operations are vector transformations, which can be represented with the transformation matrices.

Proposition 4.1: A program P with d_s local variables and assignment, summation, and swap operations is a nonlinear discrete time and input system of the form

$$\boldsymbol{x}_{n+1} = M\boldsymbol{x}_n, \text{ for } M \in U,$$

where state is a vector $\boldsymbol{x}_n \in \mathbb{R}^{d_s}$, and $M \in U \subset \mathbb{R}^{d_s \times d_s}$ is a vector transformation matrix of the state space.

The proof is in Appendix. All programs that manipulate variables are nonlinear control systems per Proposition 4.1, and the control theory tools can be applied for program analysis.

Having formulated programs as dynamical systems (X, U, D), we only need to provide a reward to formulate MDP [15], [19]. The reward is a scalar feedback on state quality. Typically, the reward is given only when the goal is reached [18], or using a simple metric of a state when available. The next section formulates the reward using sorting as an example.

B. RL Sorting Agent

In this section we develop a stable and resilient RL sorting agent. It learns once, on small arrays, and uses the learned policy to sort an array of arbitrary length. Next, we define MDP and features.

The array sorting state space is the d_s-element array itself, $\boldsymbol{x} = [x_1, \ldots, x_{d_s}]^T \in \mathbb{R}^{d_s}$. The control space is the discrete set of possible element repositions, $U = \{(i,j) | i,j = 1..d_s\}$. Action (i,j), acts as a list insert. It removes the array's i^{th} element and inserts it into the jth position. Treating arrays as vectors, the actions are permutations of the vector's coordinates, and can be represented with a transformation matrix, $M_{i,j} = [m^{i,j}_{k,l}]$, $i,j = 1 \ldots d_s$. It repositions the ith element to the jth position, $\boldsymbol{x}' = M_{i,j}\boldsymbol{x}$, when its elements are defined as

$$m^{i,j}_{k,l} = \begin{cases} 1 & k = l, (k < i \,or\, k > j) \\ 1 & (k = j, l = i) \,or\, (i \le k < j, l = k-1) \\ 0 & otherwise \end{cases}, \; i \le j \text{ or}$$

$$m^{i,j}_{k,l} = \begin{cases} 1 & k = l, (k < j \,or\, k > i) \\ 1 & (k = j, l = i) \,or\, (j < k \le i, l = k+1) \\ 0 & otherwise \end{cases}, \; i \ge j.$$

Matrices $M_{i,j}$, insert the ith element at the jth position, shift all the elements in between, and do not change elements outside the $[i, j]$ range.

The reward consists of two components: the sum of displaced adjacent elements plus a bonus for reaching a sorted array, $R(\boldsymbol{x}) = \sum_{i=2}^{d_s}(x_i - x_{i-1}) \cdot \mathrm{id}(x_i - x_{i-1} < 0) + 1000 \cdot \mathrm{id}(r_1(\boldsymbol{x}) == 0)$, where $\mathrm{id}(cond)$ equals one when the condition is true, and zero otherwise.

The state-value function approximation, V, given in (2), is a linear map of a two-dimensional feature vector. We choose features that give an advantage to first sorting areas of the array that are highly unsorted, because our goal is for the agent to perform the most with limited resources. The feature vector is two dimensional. The first feature, $\boldsymbol{F}_1$, is the number of adjacent out-of-order elements. The second feature ranks arrays with similar elements close together as more sorted than arrays with large variations between adjacent elements.

$$\boldsymbol{F}(\boldsymbol{x}) = [\boldsymbol{F}_1(\boldsymbol{x})\ \boldsymbol{F}_2(\boldsymbol{x})]^T \quad (4)$$

$$= \left[\sum_{i=2}^{d_s} \mathrm{id}(x_{i-1} > x_i)\ \sum_{i=2}^{d_s} \|x_{i-1} - x_i\|_0^2 \right]^T \quad (5)$$

where $\|x_{i-1} - x_i\|_0^2 = ((x_i - x_{i-1})^2 \mathrm{id}(x_i - x_{i-1} < 0))$. To learn V, AVI algorithm finds the parametrization $\boldsymbol{\theta}$.

Once the feature weights are learned, the RL sorting agent moves to a trajectory generation phase where it sorts arrays without further learning. Instead, at every time step, the algorithm evaluates the current array and chooses an element to move and its new position

$$(i, j) = \underset{(k,l) \in [1,..,d_s]^2}{\mathrm{argmax}}\ \boldsymbol{\theta}^T \boldsymbol{F}(M_{k,l}\boldsymbol{x}). \quad (6)$$

The chosen action, which maximizes the gain, is applied to the array. The algorithm stops when there are no more displaced elements, i.e., the array is sorted.

C. Analysis

This section analyzes RL sort. We show that the algorithm is stable, then evaluate its computational complexity and discuss its resiliency.

1) Stability Analysis: To show RL sort's stability in the absence of errors, we analyze the algorithm's monotonic progression toward the sorted array. The consequence is that the sorted array is an asymptotically stable equilibrium point of the resulting system.

Proposition 4.2: During execution of policy (6) in the absence of errors for an arbitrary array with distinct elements $\boldsymbol{x} \in \mathbb{R}^n$, and when both components of the learned weights $\boldsymbol{\theta}$ are negative ($\theta_1 < 0$, $\theta_2 < 0$), the following holds:

1) The value function $V(\boldsymbol{x})$ increases at every iteration of the algorithm, and
2) Upon termination of trajectory generation with (6), the array is sorted.

The proof, in Appendix, is based on case-by-case analysis of possible scenarios and construction of control Lyapunov function. The proof reveals that RL sort moves elements from the edges into the middle of previously sorted chains, forming increasingly longer and more dense chains.

The direct consequence of Proposition 4.2 is that if RL sort gets interrupted, the intermediate result is a more sorted array than the original one. Similarly, the impact of an erroneous comparison sets back the algorithm temporarily, but because of the MDP formulation the algorithm continues with the most current array and without expectations as to how it arrived in that state.

2) Computational Complexity of Element Moves: RL sort does not modify already sorted arrays. Thus, the lower bound on the number of element moves is $O(1)$, and computational complexity is $O(d_s^2)$. The theoretical upper bound on the number of element moves is $O(d_s^2)$. If the array has c sorted chains, then, in the worst case, there are $\lfloor \frac{d_s}{c} \rfloor$ elements in each, and the elements from the beginnings and endings of all chains are placed in the middle of a single chain, leaving the number of chains and the number of displaced adjacent pairs, $F_1(\boldsymbol{x})$, unchanged. After at most $2 * \lfloor \frac{d_s}{c} \rfloor$ element moves we are left with one less chain, and remaining chains have $\lfloor \frac{d_s}{c-1} \rfloor$ elements. Because the maximum number of chains is $c = \lfloor \frac{d_s}{2} \rfloor$, the conservative estimate of the number of element moves is $\sum_{c=1}^{\lfloor \frac{d_s}{2} \rfloor} 2 * \lfloor \frac{d_s}{c} \rfloor = O(d_s^2)$. The empirical results in Section V show that this estimate is very conservative, and that in practice the algorithm makes less element moves than Quicksort.

The computational complexity of policy (6) is $O(d_s^2)$. However, action selection with the greedy policy can be improved in several ways. First, a simple way to reduce the computational time to $O(d_s)$, is to use the knowledge gained in Table II and restrict the search to only actions that move elements from the edges to the middle of sorted chains. Second, the greedy policy can be parallelized with $O(d_s^2)$ processors, reducing its computational complexity to $O(\log d_s)$ time. With additional $O(d_s)$ storage, the policy evaluation can be done while the elements of the array are being moved. Lastly, using specialized hardware acceleration can speed up the action selection by reducing matrix multiplication to linear time, because computation is based on matrix multiplication.

3) Resiliency: When RL sort uses a faulty comparison, the assumptions of Proposition 4.2 are violated and the stability no longer holds. Thus, this section discusses RL Sort's stability in probability. Specifically, we assess the probability of failing to sort an array and monotonic progression toward the goal. We consider a p-faulty comparison component to be an id function that returns an incorrect answer with probability $0 \leq p \leq 1$, and denote it id_p. Similarly, the feature vector calculated with a p-faulty comparison is denoted $\boldsymbol{F}_p(\boldsymbol{x}) = [F_1^p(\boldsymbol{x})\ F_2^p(\boldsymbol{x})]^T$. We assume uniform random probability distribution for id function.

Proposition 4.3: Policy (6) that uses a p-faulty comparison terminates and fails to sort an array $\boldsymbol{x} \in \mathbb{R}^{d_s}$ with no further processing with probability $P = \binom{d_s}{k} p^k (1-p)^{(d_s - k)}$, where k is the number of unsorted adjacent elements $k = F_2(\boldsymbol{x})$.

The proof is in the Appendix. The consequence of Proposition 4.3 is that highly unsorted arrays are un-

likely to be recognized as sorted. The probability of terminating by mistake increases as the array becomes more sorted. It also depends on the array size; long arrays are less likely to be taken for sorted.

Next, we discuss the probability that RL sort fails to monotonically progress. Consider a partition of action set $U = G \cup N \cup W$, $G \cap N = G \cap W = N \cap W = \emptyset$.

$$G = \{(i,j) | \Delta V(\boldsymbol{x}, i, j) > 0\}, \ \|G\| = g,$$
$$N = \{(i,j) | \Delta V(\boldsymbol{x}, i, j) = 0\}, \ \|N\| = n,$$
$$W = \{(i,j) | \Delta V(\boldsymbol{x}, i, j) < 0\}, \ \|W\| = w,$$

where ${d_s}^2 = g + n + w$, and $\Delta V(\boldsymbol{x}, i, j) = V(M_{i,j}\boldsymbol{x}) < V(\boldsymbol{x})$ is the residual. Probability of choosing an action that deceases the value, $\Delta V(\boldsymbol{x}, i, j) < 0$, of the resulting array is a probability of one of the actions from $(i,j) \in W$ ending up having the biggest possible value and being selected. Let us denote p_V as the probability that the action value changes category (G, N, W) given the comparison's failure rate of p. The probability of an element from W getting the largest value is, and that value being selected is

$$\Pr((i,j) \in W | V_{p_V}(M_{i,j}\boldsymbol{x})) = \max_{(k,l) \in A} V_{p_V}(M_{k,l}\boldsymbol{x})$$
$$= \frac{w}{d_s^2} p_V \frac{1}{d_s^2} = \frac{w}{d_s^4} p_V,$$

because the probability that $V_{p_V}(M_{i,j}\boldsymbol{x})$ is the largest, and therefore selected, is d_s^{-2}.

In conclusion, small and almost sorted arrays are more likely to have setbacks while using RL sort because as the array becomes more sorted w becomes larger. We can expect to see no monotonicity violations for highly unsorted arrays, and start seeing more setbacks as the sorting progresses, a trend we see during empirical tests in Section V.

When d_s is large, it becomes unlikely that mistakes will have an important impact on the algorithm. Extensive decision-making, a downside from the computational complexity point of view, is an advantage for resiliency. A large number of actions that RL sort examines have a favorable, but not optimal, outcome. Under a favorable outcome, the algorithm selects an action that increases the value, although the increase is not maximal, therefore preserving stability. Traditional sorting algorithms generally perform one array manipulation per decision, impacting their resiliency and stability. Additionally, RL sort is more likely to make less severe mistakes as the probability increases for more sorted arrays.

V. Results

RL sort is compared to Bubble sort and Quicksort, because the two algorithms represent two sorting extremes [1]. The Bubble sort repeatedly scans the list and swaps adjacent out-of-order elements, while Quicksort selects a pivot element and creates three sublists that are smaller, equal, and larger then the pivot. Quicksort then performs a recursive sort on the lesser and greater elements and merges the three lists together. Quicksort is a single-pass algorithm making large changes in element placement. On the other hand, Bubble sort makes small changes repeatedly until it completes. The dataset consists of 100 arrays with 10 and 100 uniformly randomly drawn elements. We evaluate RL sort with error-free and 5% faulty comparison.

TABLE I
Sorting characteristics demonstrating the impact of random initial distance, the array length, and noise in the comparison routine averaged over 100 trials. Measures are the number of array element moves.

Alg.	Len.	0% Fault		5% Fault			
		# Moves		# Moves		Error	
		μ	σ	μ	σ	μ	σ
RL	10	10.6	2.6	11.3	3.0	5.3	23.1
	100	284.0	9.4	311.3	13.0	0.5	3.9
Bbbl.	10	23.1	4.9	28.0	6.7	3.1	19.2
	100	2466.4	153.2	9836.2	992.4	8.9	4.8
Quick	10	43.8	5.3	42.7	4.9	50.1	52.0
	100	846.9	63.4	816.3	42.2	255.8	74.8

A. Learning

To learn the parametrization $\boldsymbol{\theta}$, we run the AVI with discrete actions. The samples are drawn uniformly from the space of 6-element arrays with values between zero and one, $\boldsymbol{x}_s \in (0,1)^6$. The 6-element arrays provide a good balance of sorted and unsorted examples for the learning, determined empirically. We train the agent for 15 iterations. The resulting parametrization, $\boldsymbol{\theta} = [-1.4298 - 0.4216]^T$, has all negative elements and meets the conditions in Proposition 4.2.

B. Evaluation

Table III summarizes the sorting performance. RL sort finds a solution with the least changes to the array. This is because the RL sort does not make the comparisons in the same way traditional sorting algorithms do. Most of its time is spent selecting an action to perform. In the presence of a faulty comparison (Table III), the number of changes to the array that RL sort and Quicksort perform do not change significantly (less than two standard deviations). The Bubble sort, however, changes the array twice as much. We expect RL sort to seldom make severe mistakes, Quicksort does not reevaluate choices once made, and Bubble sort corrects the mistakes after additional processing. Next, we look into array error. The error is a Euclidean distance, $d(\boldsymbol{x}^o, \boldsymbol{x}^s) = \|\boldsymbol{x}^o - \boldsymbol{x}^s\|$, between an outcome of sorting with a faulty comparison, $\boldsymbol{x}_o \in \mathbb{R}^{d_s}$, and the reliably sorted array, $\boldsymbol{x}_s \in \mathbb{R}^{d_s}$. No error means that the algorithm returns a sorted array, while high error indicates big discrepancies from the sorted array. Note that this similarity metric would have been an ideal feature vector, but it is impossible to calculate

it without knowing the sorted array. With 5% fault-injection rate, the RL sort's error remains consistent and small across the array sizes, although with a relatively high standard deviation. Bubble sort has a comparable error level but makes an order of magnitude more array changes. The Quicksort completes with an order of magnitude higher error. It is clear that RL sort is resilient to noise and changes the array the least. Table III shows more comprehensive evaluation results over different datasets, with the similar conclusions.

Fig. 1 visualizes sorting progression of the same array with the three methods, in the absence of errors. Runs end at 111, 433, and 608 steps for RL sort, Quicksort, and Bubble sort, respectively. Bubble sort makes small, local changes and Quicksort's movements around the pivot make large-step movements. The RL sort (Fig. 1a) takes advantage of the structure in the data: the array is sorted into progressively larger sorted subgroups. This is because the agent reduces the number of adjacent out-of-order elements at each step. Given this, it is no surprise that RL sort needs fewer array manipulations. Figs. 1d-1c depict the same array sorted with unreliable comparison. In presence of unreliable comparisons, RL sort takes a different trajectory as the result of the faulty information, it arrives at the goal state, sorted array. The inaccurate information affects Bubble sort locally, because its decisions are local, and it too produces a sorted array. On the other hand, Quicksort, does not revisit sections of the array it previously sorted. Without the redundancy, it fails to sort the array, explaining the high error rates in Table III.

Visualizing the intermediate array values, $V(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{F}(\boldsymbol{x})$, Figs. 2a and 2b offer another view into the algorithms' progression. The RL sort with the reliable comparison monotonically progresses to the sorted array, empirically showing asymptotic stability from Proposition 4.2. Bubble sort and Quicksort have setbacks and do not progress monotonically. RL sort with faulty comparison (Fig. 2b) makes steady progress during the early phases of computation, and experiences temporary setbacks later in the processing, as the analysis in Section IV-C.3 predicted. Quicksort fails to reach the same value as RL sort because it stops computation after the first pass. Bubble sort revisits decisions and corrects the faulty decisions, and it eventually reaches the same value as the RL sort.

Lastly, the Figs. 2c and 2d evaluate resiliency based on the fault rate. Fig. 2c shows the percentage of successfully sorted arrays (out of 100) over the failure probability of the comparison routine. For fault rates of 5%, Quick sort has 0 probability of sorting an array, while Bubblesort fails 100% of the time when the fault rate is over 10%. RL Sort has a non-zero probability of sorting an array for fault rates under 50%, confirming our theoretical results. Fig. 2d measures mean and standard deviation of completion error (Euclidean distance between the terminal state and sorted array). In this graph, lower numbers are better. RL sort has consistently the lowest error. Quicksort's error is a convex, meaning that even small initial errors result in big errors in the terminal state array. Overall, RL sort is more likely to sort an array, and when it fails to sort, the array it produces will be closer to a fully sorted array, than other comparative methods.

VI. Conclusion

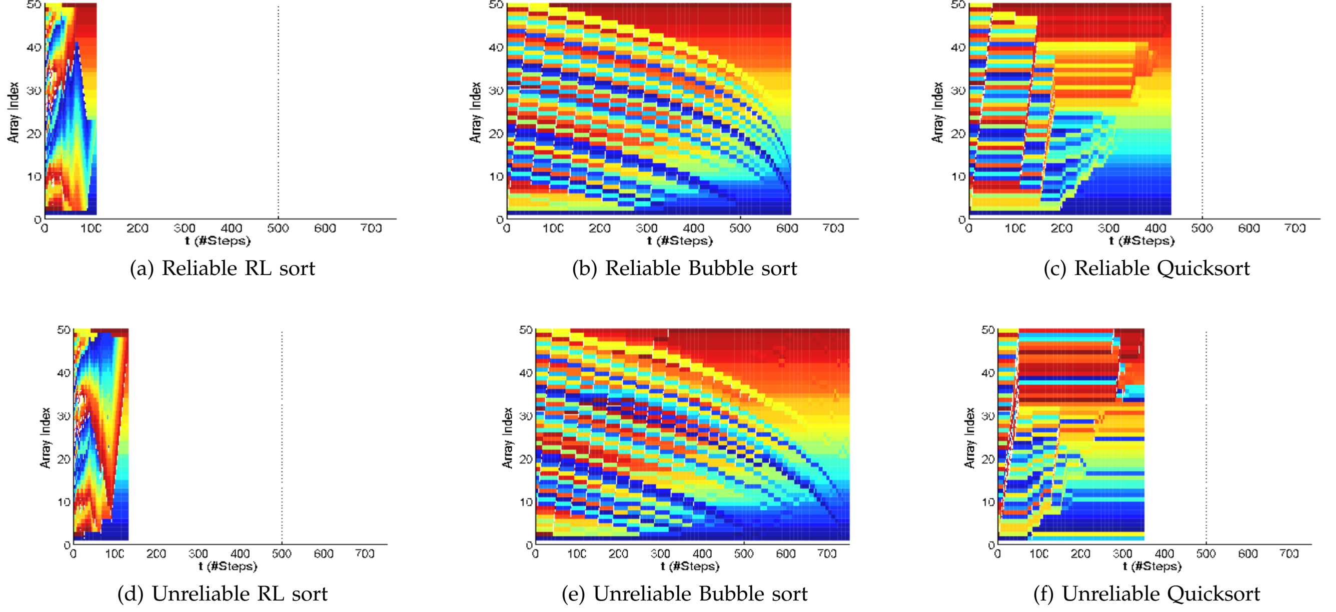
This paper presents that *stability* and *resiliency* feedback control and sequential decision making concepts address error and termination limitations of traditional computing. Treating computing as a trajectory generation problem, we apply learning methods to develop an autonomous computing agent, and use Lyapunov stability theory to analyze it. Specifically, we solve sorting with a *stable* and *resilient* sorting agent. We prove the stability in the absence of errors, and discuss the probability of success and to maintaining steady progress in presence of soft errors.

The advantages to resilient computing are that the presented method 1) makes very few assumptions about the source of the error, and 2) does not require manual programming, just a problem formulation. The high computational complexity of the resulting agent improves its resiliency in unreliable conditions. Its practicality for large-scaled general computing can be improved with use of hardware acceleration and other engineering methods.

An empirical study that compares the RL agent to two traditional sorting algorithms, confirmed the theoretical findings, and showed that the RL sorting agent completes the task with less array manipulations than even the traditional counterparts. In future work, we will develop a tighter upper bound and expected number of array manipulations for RL sort.

Acknowledgments

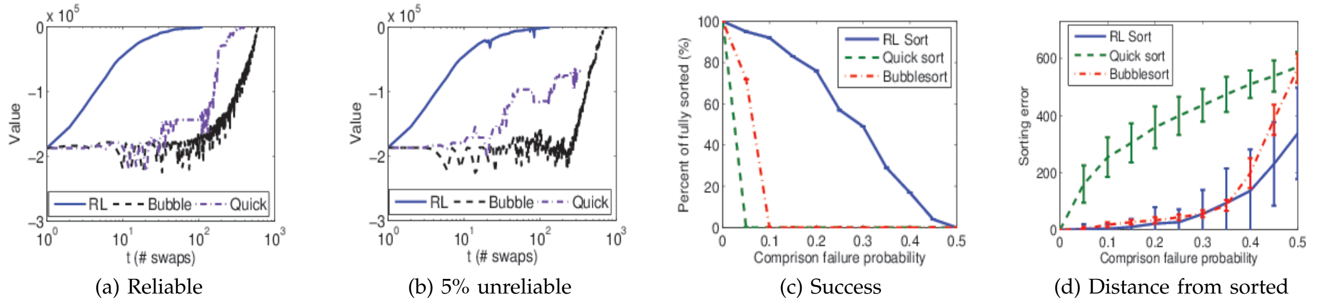
The authors thank Vikas Sindhwani, David Ackley, and Marco Morales. Tapia funded in part by the National Science Foundation under Grant Numbers IIS-1528047 and IIS-1553266. This work was supported by Sandia National Laboratories Laboratory Directed Research and Development (LDRD) Program under the Hardware Acceleration of Adaptive Neural Algorithms Grand Challenge project. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, a wholly owned subsidiary of Honeywell International, Inc., for the U. S. Department of Energys National Nuclear Security Administration under Contract de-na0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department



(a) Reliable RL sort (b) Reliable Bubble sort (c) Reliable Quicksort

(d) Unreliable RL sort (e) Unreliable Bubble sort (f) Unreliable Quicksort

Fig. 1. Sorting progression. A 50-element random array sorted with AVI, Bubble sort and Quicksort with a reliable comparison (a-c) comparison and 5% unreliable (d-f) comparison components. Time steps are on x-axis, and the array element heatmap is on y-axis. Blue colored are the smallest, and red colored are the largest array elements. Runs end when the array is fully sorted.



(a) Reliable (b) 5% unreliable (c) Success (d) Distance from sorted

Fig. 2. (a) and (b) Value progression over time for an array sorted with RL, Bubble sort, and Quicksort with a reliable (a) and 5%-faulty (b). x-axis is logarithmic. (c) and (d) Percent of successful sorting (a) and average distance from a fully sorted array, and (b) at termination after at varying comparison fault rate.

of Energy, the United States Government or National Science Foundation.

Appendix

Proof for Proposition 4.1. *Proof:* The proof is by construction. The variable manipulations are changes in the vector space, transforming one vector to another. Finding a transformation matrix between the vectors proves the proposition.

Assignment: Let $\boldsymbol{x} = [x_1, ..., x_{d_s}]^T \in \mathbb{R}^{d_s}$ be a vector representing the current state of variables, and the assignment operation $x_i \leftarrow x_j$ assigns the value of the jth variable to the ith variable. Consider a square d_s-by-d_s matrix $M^a_{i,j}$, where its elements $m^{i,j}_{k,l}, 1 \leq k, l \leq d_s$ are defined as follows:

$$m^{i,j}_{k,l} = \begin{cases} 1 & (k = i,\ l = j)\, or (k \neq i,\ k = l) \\ 0 & otherwise \end{cases}.$$

This matrix differs from the identity matrix only in the ith row, where the ith element is zero, and j^{th} is set to one. Then, vector $\boldsymbol{x}' = M^a_{i,j}\boldsymbol{x}$, has i^{th} element equal to x_j, and others unchanged. Similarly, M^c_i, where $m^i_{i,i} = c$, $m^i_{k,k} = 1$, $k \neq i$ and zero otherwise, assigns constant c to the ith variable.

Summation: We show construction of the two-variable summation action matrix. The general case can be shown with induction. Consider action matrix M^a_{i,j_1,j_2} defined with

$$m^{i,j_1,j_2}_{k,l} = \begin{cases} 1 & (k = l,\ k \neq i)\, or (k = i,\ l \in \{j_1, j_2\}) \\ 0 & otherwise \end{cases}$$

for $i, j_1, j_2 \in [1, \ldots d_s]$. As previously, this matrix differs from the identity matrix only in the ith row, where only elements j_1 and j_2 are non zero. $\boldsymbol{x}' = M^a_{i,j_1,j_2}\boldsymbol{x}$ is a vector where i^{th} element equals sum of j^{th}_1 and j^{th}_2 elements $\boldsymbol{x}$.

TABLE II
VALUE FUNCTION CHANGE ($\Delta V(M_{i,j}\boldsymbol{x})$) BASED ON POSSIBLE ACTION (i,j)

Placement of the element to be moved (i) within a sorted chain	Destination placement within a sorted chain (j)			Conclusion
	Middle $x_{j-1} < x_j$	Beginning $x_{j-1} > x_j$, $x_i < x_j < x_{j+1}$	End $x_{j-1} > x_j$, $x_{j-1} > x_i$, $x_i < x_j$	
Middle: $x_{i-1} < x_i < x_{i+1}$	N/A	N/A	N/A	Not possible
Beginning $x_{i-1} > x_i$, and $x_i < x_{i+1}$	$\theta_1(\mathrm{id}(x_{i-1} > x_{i+1}) - 1) + \theta_2(\|x_{i-1} - x_{i+1}\|_0^2 - \|x_{i-1} - x_i\|_0^2)$	N/A	N/A	> 0 because $x_i < x_{i+1}$
End: $x_{i-1} < x_i$, and $x_i > x_{i+1}$	$\theta_2(\|x_{i-1} - x_{i+1}\|_0^2 - \|x_i - x_{i+1}\|_0^2)$	N/A	N/A	> 0 because $x_{i-1} < x_i$

Element swapping: Lastly, we construct a transformation matrix $M^s_{i,j}$ that swaps x_i and x_j. Consider

$$m^{i,j}_{k,l} = \begin{cases} 1 & (k = i,\ l = j)\ or\ (k = j,\ l = i)\ or\ (k = l,\ k \neq i, j) \\ 0 & otherwise \end{cases}.$$

This matrix differs from the identity matrix in that elements (i,j) and (j,i) are ones, while the elements (i,i), and (j,j) are zero.

Finally, when the action space is set of transformation matrices, $U = \{M^a_{i,j}, M^c_i, M^a_{i,j_1,j_2}, M^s_{i,j} \mid i,j = 1,\ldots,d_s, c \in \mathbb{R}\}$, the variable space manipulation with an action $M \in U$ is a dynamical system, $\boldsymbol{x}_{n+1} = M\boldsymbol{x}_n$. ■

Proof for Theorem 4.2.

Proof: First, we show that V increases at every iteration. It is sufficient to show that the residual $\Delta V(\boldsymbol{x}, i, j) = V(M_{i,j}\boldsymbol{x}) - V(\boldsymbol{x}) > 0$ for an arbitrary unsorted array $\boldsymbol{x}$. Moving the ith element to the jth place changes the array from $\boldsymbol{x} = [x_0, \ldots, x_{j-1}, x_j, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots x_{d_s+1}]^T$ to

$$M_{i,j}\boldsymbol{x} = [x_0, \ldots, x_{j-1}, x_i, x_j, \ldots, x_{i-1}, x_{i+1}, \ldots x_{d_s+1}]^T.$$

To simplify the notation without loss of generality, we append the array with elements $\boldsymbol{x}_0$ and $\boldsymbol{x}_{d_s+1}$, such that $\boldsymbol{x}_0$ is smaller than the $\min \boldsymbol{x}$, and $\boldsymbol{x}_{d_s+1} > \max \boldsymbol{x}$. The two additional elements will not be moved during execution of policy (6), because action set U does not contain transform matrices for the two new elements.

Note that the residual $\Delta V(\boldsymbol{x}, i, j)$ depends only on neighborhood of the ith and j^{th} elements

$$\begin{aligned} \Delta V(\boldsymbol{x}, i, j) &= V(M_{i,j}\boldsymbol{x}) - V(\boldsymbol{x}) \\ &= \theta_1(\mathrm{id}(x_i > x_{j-1}) + \\ &+ \mathrm{id}(x_i > x_j) + \mathrm{id}(x_{i-1} > x_{i+1}) - \\ &- \mathrm{id}(x_{j-1} > x_j) - \mathrm{id}(x_{i-1} > x_i) - \\ &- \mathrm{id}(x_i > x_{i+1}) + \\ &+ \theta_2(\|x_i - x_{j-1}\|_0^2 + \|x_i - x_j\|_0^2 + \\ &+ \|x_{i-1} - x_{i+1}\|_0^2 - \\ &- \|x_{j-1} - x_j\|_0^2 - \|x_{i-1} - x_i\|_0^2 - \|x_i - x_{i+1}\|_0^2, \end{aligned}$$

and when the array $\boldsymbol{x}$ is not sorted $V(\boldsymbol{x}) < 0$, since $\theta_1, \theta_2 < 0$.

Consider that action (i,j) is selected at an arbitrary iteration. There are three possibilities for element x_i; x_i is in the beginning, end, or middle of a sorted chain.

End of chain: Consider x_i is at the *end* of ascending chain, $x_{i-1} < x_i$, and $x_i > x_{i+1}$. Then

$$\|x_i - x_{i+1}\|_0^2 < \|x_{i-1} - x_{i+1}\|_0^2,$$

and $\Delta V(\boldsymbol{x}, i, j)$ in the neighborhood of x_i increases (because $\theta_2 < 0$). Thus, x_i is a candidate to be selected.

Beginning of chain: Now, consider that x_i is at the *beginning* of an ascending chain,

$$x_{i-1} > x_i, \text{ and } x_i < x_{i+1}.$$

Similarly,

$$\|x_{i-1} - x_i\|_0^2 < \|x_{i-1} - x_{i+1}\|_0^2,$$

$\Delta V(\boldsymbol{x}, i, j)$ in the neighborhood of x_i increases, and x_i is a candidate to be selected.

Middle of chain: Last, consider that x_i is in the *middle* of a sorted, increasing or decreasing chain, i.e.

$$x_{i-1} < x_i < x_{i+1} \text{ or } x_{i-1} > x_i > x_{i+1}.$$

Removing x_i does not increase value function in the neighborhood of the ith element. Assume that $\boldsymbol{x}$ is not sorted, and x_i, which is in the middle of a sorted chain, is picked by the greedy policy (6). If the chain is ascending,

$$\mathrm{id}(x_{i-1} > x_i) = 0,\ \mathrm{id}(x_i > x_{i+1}) = 0,\ \mathrm{id}(x_{i-1} > x_{i+1}) = 0.$$

When the chain is decreasing,

$$\mathrm{id}(x_{i-1} > x_i) = 1,\ \mathrm{id}(x_i > x_{i+1}) = 1,$$
$$\mathrm{id}(x_{i-1} > x_{i+1}) = 1,$$
$$\|x_{i-1} - x_i\|_0^2 + \|x_i - x_{i+1}\|_0^2 < \|x_{i-1} - x_{i+1}\|_0^2.$$

Thus, removing x_i from a sorted chain does not increase $V(M_{i,j}\boldsymbol{x})$ in the neighborhood of x_i. Because $\boldsymbol{x}$ is not sorted, there is at least one element x_k such that $x_{k-1} > x_k$. Choosing either x_{k-1} or x_k will increase $\Delta V(\boldsymbol{x}, i, j)$ regardless of the direction, therefore x_i that is in the middle of a sorted chain will not be selected for the move.

Next, we look into the feasibility of the selected element's placement. Like before, the selected element

x_i, can be placed in the middle, beginning, or end of a sorted chain.

Middle of chain: Assume that x_i is moved to *middle* of an ascending chain, $x_{j-1} < x_j$. Then

$$\mathrm{id}(x_{j-1} > x_j) = \mathrm{id}(x_{j-1} > x_i) = \mathrm{id}(x_i > x_j) = 0,$$

and $\Delta V(\boldsymbol{x}, i, j)$ in the neighborhood of x_j does not change with insertion of x_i.

Beginning of chain: When x_j is the *beginning* of an ascending chain,

$$x_{j-1} > x_j, \text{ and } x_i < x_j < x_{j+1}.$$

Then,

$$\|x_{j-1} - x_j\|_0^2 > \|x_{j-1} - x_i\|_0^2 + \|x_i - x_j\|_0^2.$$

This action will not be chosen because moving into the middle of a sorted chain results in smaller $\Delta V(\boldsymbol{x}, i, j)$. Note that there are always at least two sorted chains, one at the beginning and one at the end of the array, $x_0 < x_1$, and $x_{d_s} < x_{d_s+1}$.

End of chain: When x_i is at the end of an ascending chain, using a similar argument we conclude that it will not be placed at the end of a sorted array.

Table II summarizes how the state-value function residual changes under different scenarios. A strategy that selects an element from a beginning or end of an ascending chain, and places the element into the middle of another sorted chain, results in a strictly positive change in $V(M_{i,j}\boldsymbol{x})$. For an unsorted array such element and its placement can always be found. This proves the first part of Propositions 4.2.

Proving the second part of Proposition 4.2, that the computation progresses towards a sorted array, is simple using the Lyapunov direct method. Let

$$W(\boldsymbol{y}) = -V(\boldsymbol{y} + \boldsymbol{x}^*),$$

where $\boldsymbol{x} = \boldsymbol{y} + \boldsymbol{x}^*$, and $\boldsymbol{x}^*$ is sorted $\boldsymbol{x}$.

1) $W(\mathbf{0}) = V(\boldsymbol{x}^*) = 0$

2) $W(\boldsymbol{y})$ is always positive outside of origin, because $V(\boldsymbol{x}) < 0$ for unsorted arrays.

3)

$$\begin{aligned}
\Delta W(\boldsymbol{y}) &= \\
&= -V(M_{i,j}^T(\boldsymbol{y} + \boldsymbol{x}^*)) + V(\boldsymbol{y} + \boldsymbol{x}*) \\
&= -V(M_{i,j}^T(\boldsymbol{x} - \boldsymbol{x}^* + \boldsymbol{x}^*)) + V(\boldsymbol{x} - \boldsymbol{x}^* + \boldsymbol{x}^*) \\
&= -\Delta V(\boldsymbol{x}) < 0,
\end{aligned}$$

since we showed earlier that $V(\boldsymbol{x})$ increases in subsequent stops, when $\boldsymbol{y} \neq \mathbf{0}$.

From the above, $W(\boldsymbol{y})$ is a control Lyapunov function, and $\boldsymbol{x}^*$ is an asymptotically stable equilibrium. Consequently, the computation with respect to the control policy (6) will ensure that the computation progresses to the sorted array starting at an arbitrary initial array. This proves the total correctness part of the proposition, and concludes the proof. ■

Proof for Proposition 4.3. *Proof:* Trajectory generation terminates for an unsorted array $\boldsymbol{x}$, only if faulty comparison causes $\boldsymbol{\theta}^T\boldsymbol{F}(\boldsymbol{x})$ to evaluate as 0 in (6). The probability of $\boldsymbol{F}(\boldsymbol{x})$ evaluating as 0 is if all calls to $\mathrm{id}(x_{i-1} > x_i), i = 2, \ldots, d_s$ return 0. Since there are k adjacent elements that are displaced in $\boldsymbol{x}$ the probability

$$\begin{aligned}
&\Pr(\boldsymbol{F}_p(\boldsymbol{x}) = 0) = \\
&= \prod_{i=2}^{d_s} (\Pr(\mathrm{id}_p(x_{i-1} \leq x_i) | \mathrm{id}(x_{i-1} > x_i)) \\
&\cdot \Pr(\mathrm{id}_p(x_{i-1} \leq x_i) | \mathrm{id}(x_{i-1} \leq x_i))) \\
&= \binom{d_s}{k} p^k (1-p)^{(d_s - k)},
\end{aligned}$$

because there are k unsorted adjacent elements. ■

References

[1] D. H. Ackley. Beyond efficiency. *Commun. ACM*, 56(10):38–40, Oct. 2013.

[2] M. Ajtai, V. Feldman, A. Hassidim, and J. Nelson. Sorting and selection with imprecise comparisons. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I*, ICALP '09, pages 37–48, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] K. J. Astrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Apr. 2008.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecure in Practice, 2nd ed.* Addison Wesley, 2003.

[5] H. Bou-Ammar, H. Voos, and W. Ertel. Controller design for quadrotor uavs using reinforcement learning. In *IEEE International Conference on Control Applications (CCA)*, pages 2130–2135, 2010.

[6] L. Buşoniu, R. Babuška, B. De Schutter, and D. Ernst. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press, Boca Raton, Florida, 2010.

[7] S. Caminiti, I. Finocchi, and E. G. Fusco. Local dependency dynamic programming in the presence of memory faults. In *In STACS, volume 9 of LIPIcs*, pages 45–56, 2011.

[8] D. Ernst, M. Glavic, P. Geurts, and L. Wehenkel. Approximate value iteration in the reinforcement learning context. application to electrical power system control. *International Journal of Emerging Electric Power Systems*, 3(1):1066.1–1066.37, 2005.

[9] A. Faust, I. Palunko, P. Cruz, R. Fierro, and L. Tapia. Automated aerial suspended cargo delivery through reinforcement learning. *Artif. Intell.*, 247:381 – 398, 2017. Special Issue on AI and Robotics.

[10] A. Faust, P. Ruymgaart, M. Salman, R. Fierro, and L. Tapia. Continuous action reinforcement learning for control-affine systems with unknown dynamics. *Acta Automatica Sinica, in press*, 2014.

[11] I. Finocchi and G. F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332, Oct. 2008.

[12] D. Hamlet and J. Maybee. *The Engineering of Software*. Addison Wesley, 2001.

[13] T. B. Jones and D. H. Ackley. Comparison criticality in sorting algorithms. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 726–731. IEEE, 2014.

[14] K. E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, USA, April 1993. IEEE Press.

[15] F. L. Lewis, D. Vrabie, and K. G. Vamvoudakis. Reinforcement learning and feedback control: Using natural decision methods to design optimal adaptive controllers. *IEEE Control Systems Magazine*, 32(6):76–105, Dec 2012.

[16] C. Moore and S. Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[17] T. J. Perkins and A. G. Barto. Lyapunov design for safe reinforcement learning. *The Journal of Machine Learning Research*, 3:803–832, 2003.

TABLE III

EXPENDED SORTING CHARACTERISTICS DEMONSTRATING THE IMPACT OF RANDOM INITIAL DISTANCE, THE ARRAY LENGTH, AND NOISE IN THE COMPARISON ROUTINE AVERAGED OVER 100 TRIALS. MEASURES THE NUMBER OF ARRAY ELEMENT MOVES. THE RESULTS COVER SELECTION SORT, IN ADDITION TO BUBBLE AND QUICKSORT. THE SELECTION SORT, SELECTS THE MINIMUM AND PLACES IT AT THE BEGINNING OF THE ARRAY. THE TABLE PRESENTS THE RESULTS FOR DIFFERENT SORTED INITIAL CONDITIONS: FULLY SORTED, REVERSE SORTED ARRAYS, SORTED WITH A GAUSSIAN DISPLACEMENT, AND UNIFORMLY RANDOMLY SHUFFLED ARRAYS, OVER ARRAYS OF 5, 10, 50, 100 ELEMENTS LONG. THE RESULTS ARE CONSISTENT ACROSS THE EVALUATIONS.

Algorithm	Dataset	Array length	Reliable comparison # Moves μ	Reliable comparison # Moves σ	5% Faulty comparison # Moves μ	5% Faulty comparison # Moves σ	5% Faulty comparison Error μ	5% Faulty comparison Error σ
RL sort	Sorted	5	1.00	0.00	1.00	0.00	2.43	13.02
		10	1.00	0.00	1.00	0.00	2.12	15.20
		50	1.00	0.00	1.00	0.00	6.22	26.57
		100	1.00	0.00	1.00	0.00	4.82	23.27
	Reversed	5	5.00	0.00	5.23	0.63	2.30	20.01
		10	10.00	0.00	10.83	1.33	4.99	20.70
		50	50.00	0.00	55.88	3.46	0.36	3.57
		100	100.00	0.00	110.31	7.00	3.62	25.32
	Gaussian	5	3.54	0.91	3.74	1.09	10.12	71.90
		10	9.83	2.34	10.41	2.76	23.52	111.01
		50	109.45	6.21	119.35	8.14	20.58	101.99
		100	273.47	9.71	298.01	12.12	25.91	131.13
	Random	5	3.49	1.05	3.79	1.43	2.35	20.02
		10	10.66	2.69	11.34	3.05	5.31	23.16
		50	112.79	7.33	123.59	9.21	6.11	26.02
		100	284.02	9.42	311.38	13.00	0.57	3.97
Selection	Sorted	5	0.00	0.00	0.43	0.62	38.08	59.78
		10	0.00	0.00	1.83	1.04	117.26	66.88
		50	0.00	0.00	31.25	3.05	597.80	56.18
		100	0.00	0.00	79.73	3.16	1009.42	46.09
	Reversed	5	4.00	0.00	3.93	0.26	10.60	28.41
		10	9.00	0.00	8.92	0.27	10.96	21.41
		50	49.00	0.00	48.99	0.10	9.42	7.77
		100	99.00	0.00	98.94	0.24	8.19	4.59
	Gaussian	5	2.67	0.87	2.73	0.92	110.38	179.76
		10	7.09	1.33	7.19	1.24	231.80	204.31
		50	45.35	1.82	45.73	1.72	891.94	228.69
		100	94.50	2.02	95.31	1.88	1802.33	211.98
	Random	5	2.82	0.94	2.90	0.92	23.10	44.82
		10	7.15	1.04	7.34	1.00	53.93	59.19
		50	45.72	1.66	45.99	1.65	260.58	65.86
		100	95.04	1.82	95.76	1.60	513.70	63.03
Bubble	Sorted	5	0.00	0.00	0.52	1.29	0.00	0.00
		10	0.00	0.00	1.96	3.40	0.50	3.53
		50	0.00	0.00	714.94	824.94	0.66	2.68
		100	0.00	0.00	9331.80	2369.31	8.03	4.39
	Reversed	5	10.00	0.00	11.02	2.22	3.00	24.10
		10	45.00	0.00	50.93	4.85	3.42	14.57
		50	1225.00	0.00	2058.29	605.55	1.06	4.03
		100	4950.00	0.00	9980.28	195.54	9.97	4.71
	Gaussian	5	5.05	2.20	6.03	3.01	31.54	143.56
		10	21.61	5.06	27.27	8.29	8.79	43.61
		50	615.51	57.93	1443.22	678.96	2.01	7.51
		100	2461.76	144.96	9895.62	708.73	51.65	45.68
	Random	5	5.02	2.24	5.85	2.96	6.84	26.37
		10	23.19	4.95	28.08	6.73	3.13	19.21
		50	609.82	58.48	1517.21	815.87	0.91	3.34
		100	2466.44	153.20	9836.22	992.49	8.96	4.84
Quicksort	Sorted	5	16.62	2.60	16.90	2.79	26.55	48.17
		10	42.05	4.67	43.19	5.29	53.83	60.25
		50	358.57	25.43	343.96	21.16	176.65	61.70
		100	843.57	63.64	810.67	44.09	245.37	78.74
	Reversed	5	16.46	2.43	16.37	3.01	31.13	53.62
		10	42.45	4.83	43.11	5.44	60.17	53.58
		50	356.89	25.17	350.91	23.44	173.83	66.61
		100	846.15	50.54	826.82	42.72	261.82	77.31
	Gaussian	5	16.50	3.00	17.02	3.22	74.46	143.95
		10	42.57	5.03	42.97	4.78	181.86	207.35
		50	352.08	26.16	349.03	22.86	663.98	273.06
		100	854.65	68.18	812.52	44.76	883.20	233.68
	Random	5	16.16	2.44	16.87	3.13	30.96	51.06
		10	43.83	5.32	42.77	4.92	50.13	52.02
		50	358.22	25.25	348.83	19.99	181.51	60.64
		100	846.99	63.46	816.34	42.29	255.82	74.84

[18] R. Sutton and A. Barto. *A Reinforcement Learning: an Introduction*. MIT Press, MIT, 1998.

[19] R. S. Sutton, A. G. Barto, and R. J. Williams. Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems Magazine*, 12(2):19–22, April 1992.

[20] S. White, T. Martinez, and G. Rudolph. Automatic algorithm development using new reinforcement programming techniques. *Computational Intelligence*, 28(2):176–208, 2012.