Algorithm Selection using Reinforcement Learning

Michail G. Lagoudakis MGL@CS.DUKE.EDU
Department of Computer Science, Duke University, Durham, NC 27708, USA

Michael L. Littman MLITTMAN@RESEARCH.ATT.COM
Shannon Laboratory, AT&T Labs – Research, Florham Park, NJ 07932, USA
Department of Computer Science, Duke University, Durham, NC 27708, USA

Abstract

Many computational problems can be solved by multiple algorithms, with different algorithms fastest for different problem sizes, input distributions, and hardware characteristics. We consider the problem of *algorithm selection*: dynamically choose an algorithm to attack an instance of a problem with the goal of minimizing the overall execution time. We formulate the problem as a kind of Markov decision process (MDP), and use ideas from reinforcement learning to solve it. This paper introduces a kind of MDP that models the algorithm selection problem by allowing multiple state transitions. The well known Q-learning algorithm is adapted for this case in a way that combines both Monte-Carlo and Temporal Difference methods. Also, this work uses, and extends in a way to control problems, the Least-Squares Temporal Difference algorithm (LSTD(0)) of Boyan. The experimental study focuses on the classic problems of order statistic selection and sorting. The encouraging results reveal the potential of applying learning methods to traditional computational problems.

1. Introduction

When performing a repetitive task, people often find ways of optimizing their behavior to make it faster, cheaper, safer, or more reliable. Computer systems execute tasks that are far more repetitive and could benefit considerably from optimization. Programmers and source-level compilers work hard to reorganize computations to make them more efficient, but as computer systems become more complex and "mobile" programs are expected to run efficiently on a wide variety of hardware platforms, squeezing maximum performance out of a program requires run-time information.

A challenging research goal is to design a run-time system that can repeatedly execute a program, learning over time to make decisions that speed up the overall execution time. Since the right decisions may depend on the problem size and parameters, the machine characteristics and load, the data distribution, and other uncertain factors, this can be quite challenging. As a first attempt, we attack the following *algorithm selection* problem. We require that the programmer provide (a) a set of algorithms that are equivalent in terms of the problem they solve, but can differ in, for example, how their running time scales with problem size, and (b) a set of instance features, such as problem size, that can be used to select the most appropriate algorithm from the set for a given problem instance. We show how a reinforcement learning approach can be used to select the right algorithm for each instance at run-time based on the instance features.

Recall that a *recursive algorithm* is one that solves a problem by doing some preprocessing to reduce the input problem to one or more subproblems from the same class, solves the subproblems, then performs some postprocessing to turn the solutions to the subproblems into a solution for the original problem. Because each of the subproblems generated by a recursive algorithm belongs to the same class as the original problem, each gives rise to a new algorithm selection problem. Thus, when recursive algorithms are included in the algorithm set, the algorithm selection problem becomes a sequential decision problem.

Related work (Lobjois & Lemaître, 1998; Fink, 1998) treats algorithms in a black-box manner: each time a single algorithm is selected and applied to the given instance. Our focus is on algorithm selection *while* the instance is being solved. In that sense, each instance is solved by a mixture of algorithms formed dynamically at run-time.

The remainder of this section develops a simple example to clarify the definition of the algorithm selection problem. Section 2 connects the problem to that of solving a Markov decision process and Section 3 explains how a learning al-

gorithm can be applied to improve performance. Section 4 discusses approximation methods for the value function, and, finally, Section 5 provides results for two initial studies using the problems of order statistic selection and sorting.

As a simple concrete example, let's consider creating a system for sorting. We write two algorithms: shellsort and bubblesort. Shellsort has a bit more overhead, and thus can run a bit more slowly for small problems. However, its asymptotic running time for a list of n items is $O(n^{3/2})$ in contrast to bubblesort's $O(n^2)$, so we'd expect shellsort to be preferable for large problems. If we use only problem size, n, to decide which algorithm to run, the algorithm selection problem reduces to finding an optimal cutoff n' such that we sort lists of fewer than n' items with bubblesort and longer lists with shellsort.

Now, consider adding mergesort to our algorithm set. Mergesort is an $O(n \log n)$ recursive algorithm. It takes a list of n items, separates it into two lists of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, sorts them individually, and finally combines the two small sorted lists into a single sorted list. Since mergesort is the most efficient algorithm in the set for large lists, a large list will be sorted by applying mergesort repeatedly until the resulting subproblems are sufficiently small. At this point, either shellsort or bubblesort should be applied.

2. Algorithm Selection as an MDP

The algorithm selection problem can be encoded as a kind of Markov decision process (Puterman, 1994) (MDP) consisting of states, actions, costs, transitions, and an objective function. The state of the MDP is represented by the current instantiation of the instance features. To fully satisfy the Markov property, some unknown factors, like data distribution and machine characteristics, should be part of the process' state. However, such information is not only unavailable, but would also make the state space extremely large and perhaps overly expensive to manipulate on the fly. We treat such factors as unmodeled hidden state and assume their influence is negligible.

Actions are the different algorithms that can be selected. Non-recursive algorithms are terminal in that they are not followed by a state transition and the corresponding process terminates. In contrast, recursive algorithms cause transitions to other states, which correspond to the subproblems created by the recursive algorithm. These state transitions are non-deterministic in general, especially when randomization is used as part of the recursive algorithm.

The immediate cost for choosing some algorithm (action) on some problem (state) is precisely the real time taken for that execution, excluding any time taken in recursive calls. The total (undiscounted) cost accumulated while fully solving a problem is exactly the total time taken to solve the problem (see Figure 1). The objective is to find a policy, a mapping from values of instance features to algorithms, such that the expected total execution time is minimized. For a fixed policy, the value of a state s is the expected time to solve a problem described by state s using the algorithms selected by the policy. Note that the cost function is unknown and non-deterministic in general, since it may depend on several uncertain and hidden factors.

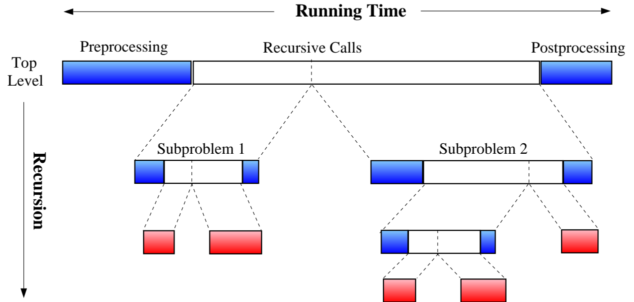

Figure 1. For each (sub)problem the shaded part of the running time indicates the immediate cost.

State transitions are a bit more complex in the case of recursive algorithms. From the MDP point of view, the multiple subproblems that are created and solved by a recursive algorithm result in transitions to multiple states—this violates the standard MDP definition. For example, mergesort divides the input to be sorted into two pieces, each corresponding to a different state, yielding a 1–to–2 state transition. However, as long as a sequential model of computation is used, we can safely treat each of these transitions to a new state independently, and the total cost will be the sum of the individual total costs for each subproblem. One can think of it as cloning the MDP and generating one copy for each transition.

There is a strong relation between the recurrence equations used to analyze the running time of recursive algorithms and the Bellman equation for the algorithm selection problem. The standard recurrence for mergesort is (Cormen et al., 1990)

$$T(n) = 2T(n/2) + \Theta(n), \quad T(1) = \Theta(1),$$

where $T(n)$ represents the running time on an instance of size n. The Bellman equation for the state value function of the Markov chain underlying mergesort is

$$V(s_n) = 2V(s_{n/2}) + R(s_n, a_m), \quad V(s_1) = 0,$$

where $R(s_n, a_m)$ is the cost for choosing mergesort in state s_n that corresponds to an instance of size[1] n; the Bellman

[1]Size is the most crucial instance feature for most problems. In presenting our method we assume that the state of the MDP

equation captures the underlying structure of the recursive algorithm.

In the most general case, the average running time $T(n)$ of a recursive algorithm that creates k subproblems of sizes $n_1, n_2, ..., n_k$, is described by the recurrence

$$T(n) = E\left[\sum_{j=1}^{k} T(n_j) + t(n)\right],$$

where $t(n)$ is the preprocessing and postprocessing time. On the other hand, the value of a state s_n under a fixed deterministic policy would be expressed as follows:

$$V(s_n) = E\left[\sum_{j=1}^{k_a} V(s_{n_j}) + R(s_n, a)\right],$$

where a is the algorithm chosen by the policy for state s_n, s_{n_j} are the states describing the resulting subproblems, and $R(s_n, a)$ is the cost for choosing a in state s_n. Although $T(n)$ corresponds to $V(s_n)$, it is expected that $V(s_n) < T(n)$, that is, the expected time for the combined algorithm is less than the time for the recursive algorithm alone.

In general, there is no model of the MDP available and thus, in order to act optimally, either a model must be learned by experience, or a model-free approach must be used. We choose the second track and focus on learning the state–action value function $Q(s, a)$. In this case, the Bellman optimality equation becomes

$$Q(s_n, a) = E\left[\sum_{j=1}^{k_a} \min_{a'}\{Q(s_{n_j}, a')\} + R(s_n, a)\right].$$

3. Learning Mechanism

Our learning mechanism is a variation of the well known Q-learning algorithm (Watkins & Dayan, 1992), adapted to account for multiple state transitions. The general (undiscounted) update equation of Q-learning is:

$$Q^{(t+1)}(s_t, a_t) = (1-\alpha)Q^{(t)}(s_t, a_t) + \alpha\left[R_{t+1} + \min_a\left\{Q^{(t)}(s_{t+1}, a)\right\}\right],$$

where s_t is the state at time t, a_t is the action taken at time t, R_{t+1} is the one-step cost for that decision, and α is the learning rate. If a_t is a non-recursive algorithm, the resulting state is terminal and has a cost of 0, so the update rule, reduces to

$$Q^{(t+1)}(s_t, a_t) = (1-\alpha)Q^{(t)}(s_t, a_t) + \alpha R(s_t, a_t).$$

consists of solely the instance size, but, in general, several other features may be used.

For recursive algorithms the learning rule is a little more involved. For the sake of simplicity, let's consider a recursive algorithm that generates only two subproblems (generalization to more subproblems is easy). In this case, the Q-learning rule is

$$Q^{(t+1)}(s_t, a_t) = (1-\alpha)Q^{(t)}(s_t, a_t) + \alpha\left[R(s_t, a_t) + \min_a\left\{Q^{(t)}(s_1, a)\right\} + \min_a\left\{Q^{(t)}(s_2, a)\right\}\right],$$

where s_1 and s_2 are the states (at time $t+1$) corresponding to the two subproblems. Notice that the target value depends on two estimates, which can introduce significant bias depending on the accuracy of the value function. In addition, multiple bootstrapping can easily cause divergence of the value function to wrong estimates if a function approximator is used (Boyan & Moore, 1995). The two resulting states must be visited individually in turn, as both subproblems must be solved. That means that it is necessary to store state information for all the pending states along the current path in the recursion tree.

The update rule above makes use of previous estimates in updating the value of the current state–action pair in the spirit of Temporal Difference (TD) algorithms (Sutton & Barto, 1998). Alternatively, one could "unfold" (solve completely) each of the two subproblems, adding the individual costs at each step. This is the "Monte-Carlo return", $\mathfrak{R}_\pi(s) = \sum_t R(s_t, a_t)$, and expresses the sum of all individual costs when starting with a subproblem corresponding to state s and following the policy π until the subproblem has been fully solved. To get good estimates, the policy π should not take any exploratory actions. Typically, π is the greedy policy with respect to the current value function. Although $\mathfrak{R}_\pi(s)$ is an unbiased estimate of the target value of $Q(s, a)$, it has high variance as it depends on several returns and is not available before the end of the episode. Unfolding both subproblems would result in a pure Monte-Carlo (MC) algorithm with the following update rule and the shortcomings just mentioned:

$$Q^{(t+1)}(s_t, a_t) = (1-\alpha)Q^{(t)}(s_t, a_t) + \alpha\left[R(s_t, a_t) + \mathfrak{R}_\pi(s_1) + \mathfrak{R}_\pi(s_2)\right].$$

Our learning rule combines the TD and MC rules above, by taking the MC approach on one subproblem and the TD approach on the other. In other words, one subproblem (say, the smallest one) is unfolded and its "Monte-Carlo return" is added to the current one–step return, before bootstrapping and recursing on the other. This is a viable alternative in this problem because of the one–to–many state transitions. The update rule takes the form:

$$Q^{(t+1)}(s_t, a_t) = (1-\alpha)Q^{(t)}(s_t, a_t) + \alpha\left[\underbrace{R(s_t, a_t) + \mathfrak{R}_\pi(s_1)}_{R_{t+1}} + \min_a\left\{Q^{(t)}(s_2, a)\right\}\right],$$

where s_2 is the state corresponding to the subproblem we recurse on. By choosing s_2 to be the largest one (or the "hardest" to solve, in general), we achieve several things: (1) more opportunities for later exploration, (2) less variance in R_{t+1}, and (3) small recursion stack (for unfolding the small subproblem). In addition, our problem becomes an ordinary MDP with single state transitions, with the extra transition effectively pushed into the cost function. Also, unlike the pure TD approach, there is no need for extra state information storage. Figure 2 clarifies all these issues. This learning rule is used in all our experiments.

An issue related to our learning rule concerns the availability of $R(s, a)$ during learning. If the last step of the recursive algorithm is one or more recursive calls, then $R(s, a)$ is immediately available before any attempt to solve the subproblems is made. Thus, the system can learn about the current state by immediately applying the learning rule and then continuing independently with the subproblems discarding the current state information. This is similar to the use of "tail recursion" to improve the efficiency of recursive calls. However, if the algorithm requires some amount of postprocessing work after one or more subproblems are solved, the return $R(s, a)$ is delayed until these subproblems have been captured. Clearly, learning is delayed in this case and state information storage is necessary. In our experiments in Section 5, we take advantage of the "tail recursion" as this is allowed by the algorithms we explored.

4. Generalization and Approximation

In this initial study, we have used both table-based and approximation methods to represent the value function and cope with the size of the state space. In particular, we make use of state aggregation and linear architectures.

State aggregation is primarily used to compress specific instance features, like problem size. The rationale is that although the running time of an algorithm might be significantly different for small feature values, this relative difference fades out as values become large. For example, sorting 20 elements is relatively more expensive than sorting 10 elements, but there is almost no relative difference between sorting 5020 and 5010 elements. So, in order to avoid an "explosion" of the state space, in our experiments we use logarithmic compression that allows for high resolution at small feature values and progressively lower resolution as values grow. In particular, the value v of an instance feature is mapped to v', according to $v' = \lceil \log_{1.1}(v + 1) \rceil$. This formula[2] maps 100, 1000, and 10000 to 49, 73, and 97 respectively.

[2]The base 1.1 of the logarithm is an empirically-derived value that simply provides the desired resolution. The unit increment is used to overcome states values of 0.

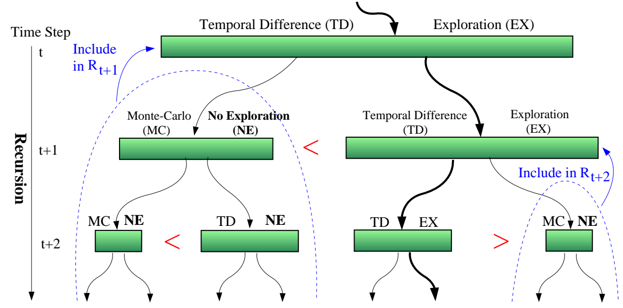

Figure 2. The learning mechanism. The Monte-Carlo return from the smaller subproblem is included in the cost of its parent, followed by a transition to the bigger subproblem. The same pattern applies recursively at all levels/time steps. Notice that once exploration is prevented at some node, it is prevented in the whole subtree under the node. The bold arrows show the trajectory of the standard MDP.

Linear architectures are used to approximate the value function. Recall that such an approximator approximates $Q(s, a)$ as a linear combination $\boldsymbol{\phi}(s, a)^\mathsf{T}\mathbf{w}$ of k basis functions $\boldsymbol{\phi}(s, a)$ with coefficients (or weights) $\mathbf{w}$. The k weights $\mathbf{w}$ are estimated in a way that minimizes discrepancy with the observed data in the least-squares sense. The observed data take the form $\{s_t, a_t, Q^{(t+1)}(s_t, a_t)\}$ for $t = 1, 2, ...$, where $Q^{(t+1)}(s_t, a_t)$ is the new (updated) value given by our learning rule in Section 3. Ideally, we would like $\boldsymbol{\phi}(s_t, a_t)^\mathsf{T}\mathbf{w} = Q^{(t+1)}(s_t, a_t)$ to be true for all data. Using $\mathbf{\Phi}$ to denote the matrix with rows $\boldsymbol{\phi}(s_t, a_t)^\mathsf{T}$ and $\mathbf{q}$ to denote the vector with components $Q^{(t+1)}(s_t, a_t)$, the least-squares solution for $\mathbf{w}$ is given by solving the $k \times k$ linear system

$$(\mathbf{\Phi}^\mathsf{T}\mathbf{\Phi})\mathbf{w} = \mathbf{\Phi}^\mathsf{T}\mathbf{q} \implies \mathbf{w} = (\mathbf{\Phi}^\mathsf{T}\mathbf{\Phi})^{-1}\mathbf{\Phi}^\mathsf{T}\mathbf{q}.$$

The matrix $\mathbf{\Phi}$ and the vector $\mathbf{q}$ can become extremely big as data accumulate. Fortunately, we need only maintain the $k \times k$ matrix $\mathbf{A} = \mathbf{\Phi}^\mathsf{T}\mathbf{\Phi}$ and the k-dimensional vector $\mathbf{b} = \mathbf{\Phi}^\mathsf{T}\mathbf{q}$, which can be incrementally updated with new data as follows:

$$\begin{aligned}
\mathbf{A}^{(t+1)} &= \begin{pmatrix} \mathbf{\Phi}^\mathsf{T} & \boldsymbol{\phi}(s_{t+1}, a_{t+1}) \end{pmatrix} \begin{pmatrix} \mathbf{\Phi} \\ \boldsymbol{\phi}(s_{t+1}, a_{t+1})^\mathsf{T} \end{pmatrix} \\
&= \mathbf{\Phi}^\mathsf{T}\mathbf{\Phi} + \boldsymbol{\phi}(s_{t+1}, a_{t+1})\boldsymbol{\phi}(s_{t+1}, a_{t+1})^\mathsf{T} \\
&= \mathbf{A}^{(t)} + \boldsymbol{\phi}(s_{t+1}, a_{t+1})\boldsymbol{\phi}(s_{t+1}, a_{t+1})^\mathsf{T} ,
\end{aligned}$$

$$\begin{aligned}
\mathbf{b}^{(t+1)} &= \begin{pmatrix} \mathbf{\Phi}^\mathsf{T} & \boldsymbol{\phi}(s_{t+1}, a_{t+1}) \end{pmatrix} \begin{pmatrix} \mathbf{q} \\ Q^{(t+2)}(s_{t+1}, a_{t+1}) \end{pmatrix} \\
&= \mathbf{\Phi}^\mathsf{T}\mathbf{q} + \boldsymbol{\phi}(s_{t+1}, a_{t+1})Q^{(t+2)}(s_{t+1}, a_{t+1}) \\
&= \mathbf{b}^{(t)} + \boldsymbol{\phi}(s_{t+1}, a_{t+1})Q^{(t+2)}(s_{t+1}, a_{t+1}).
\end{aligned}$$

The weights can be updated by $\mathbf{w}^{(t)} = \left(\mathbf{A}^{(t)}\right)^{-1}\mathbf{b}^{(t)}$ whenever needed. In this work, we use a separate linear architecture for each algorithm a, each one having its own set of weights, that is $\mathbf{w} = \mathbf{w}(a)$.

This least-squares approach is similar to the one described by Boyan (1999), and actually extends the LSTD(λ) algorithm to general MDPs for $\lambda = 0$.

5. Results

We have applied the ideas above on two fundamental computational problems: order statistic selection and sorting. These early experimental results[3] reveal that there is potential for getting the most out of well-known algorithms by combining them as suggested in the previous sections.

5.1 Order Statistic Selection

For the *order statistic selection problem*, we are given an array of n (unordered) numbers and some integer index i, $1 \le i \le n$. We would like to select the number that would rank i-th in the array if the numbers were sorted in ascending order. There are several algorithms for order statistic selection. We picked two of them such that neither is best in all cases, otherwise learning would not really help[4].

DETERMINISTIC SELECT (Cormen et al., 1990) is a recursive worst case linear time algorithm. It finds a good partitioning element by making a recursive call to find the median of a subset of the input. That subset consists of the medians of every five elements of the input, and therefore its size is a fifth of the original size. Then, the original input is partitioned and a recursive call is made to the appropriate (left or right) subproblem. The size of this subproblem varies, but it is no less than $3n/10 - 6$ and no more than $7n/10+6$, if n is the original size. Hence, two subproblems are solved at each recursive call. The recursion continues until the desired element is restricted in a subset of size less than or equal to 5 from where it can be easily isolated. The performance of the algorithm is almost invariant with respect to the value of the index (assuming fixed array size).

HEAP SELECT is a (non-recursive) algorithm with $O(n \log n)$ worst case running time. The basic step of this algorithm is the construction of a binary heap between the position i and the closest end of the array. Without loss of generality, assume that i is closer to the left end, i.e., $i \le n/2$ (the other case is symmetric). All the elements between positions 1 and i are organized into a heap, whose root is located at position i and holds the maximum element. Then, the algorithm iterates through the remaining elements; if an element is smaller than the root element of the heap, the two elements are exchanged and the new root is pushed into the heap to maintain the heap property. At the end, the desired element is located at the root of the heap. To see this, notice that all elements in the heap are smaller than or equal to the elements outside the heap. Obviously, the closer the index to the left end, the smaller the heap and the faster the algorithm, since $T(n,i) = \Theta(i) + O\left((n-i)\log i\right)$ for $i \le n/2$.

Figure 3 (in addition to other information) shows the average running time of the two algorithms for randomly generated instances of fixed size (10000) and varying index $(1 - 10000)$. As expected, HEAP SELECT performs much better than DETERMINISTIC SELECT for indices close to the ends. However, for indices close to the middle (e.g. medians) DETERMINISTIC SELECT outperforms HEAP SELECT. A similar picture holds for other array sizes as well. Thus, there is potential for a better average running time if the two algorithms are combined.

As a first attempt to learn how to combine the two algorithms, we used a tabular approach. The state of the process, in this case, consists of two instance features, namely the size of the input n and the distance d of the index from the closest end of the array ($d = \min\{i, n-i+1\}$). We assume that the problem is symmetric with respect to the middle of the array to reduce the range of d: selecting, say, the 10th element is equivalent to selecting the 91st one out of 100 elements. Also, the difference between selecting the 4000th element among 10000 elements and the 4010th element in 9995 elements is so small that discriminating between these two cases is not of much help. So, in order to avoid an explosion of the state space, we logarithmically compress the two features as described in Section 4. Given that, half a table of size 90×83 is sufficient to represent the value function.

We trained the system on thousands of randomly generated instances ($\approx 100,000$) of fixed size (10000) and varying index $(1 - 10000)$. To facilitate training, we first trained on several instances of smaller size. A $1-\epsilon$ policy with high degree of exploration ($\epsilon = 0.6$) was used during training. Two decreasing learning rates were used, one for DETERMINISTIC SELECT ($\alpha_1 = 0.4$ initially) and one for HEAP SELECT ($\alpha_2 = 0.7$ initially). DETERMINISTIC SELECT has varying cost for a given state because of the non–deterministic transitions, whereas HEAP SELECT is quite invariant. This difference is reflected in the two learning rates. The results are shown in Figure 3.

The "cut-off point algorithm" selects HEAP SELECT when the index is within the first 13% or the last 7% of the input, and DETERMINISTIC SELECT otherwise. The two cut-off

[3] All experiments were performed on a Sun Ultra 5 machine using MATLAB code. All running time plots represent averages of 10 runs per data point. Learning was turned off during performance testing.

[4] We excluded RANDOMIZED SELECT because it was consistently best in our initial studies.

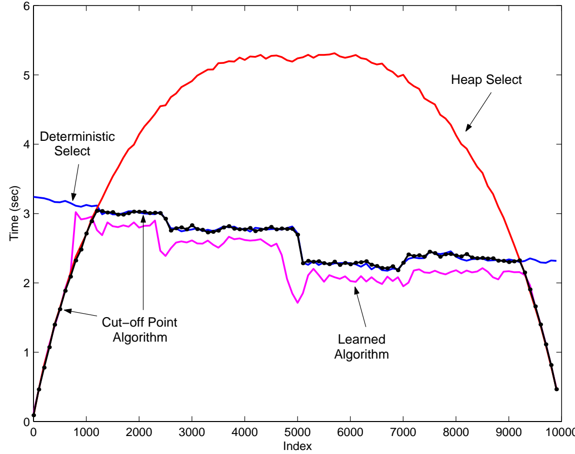

Figure 3. Results for order statistic selection (tabular case).

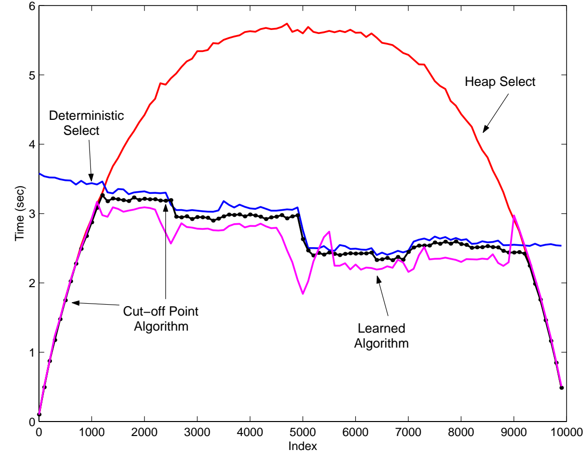

Figure 4. Order statistic selection (linear architecture).

points were determined directly from the crossover points in Figure 3. Thus, it implements an empirically derived policy, typical of those found in optimized software implementations. The learned algorithm, however, performs better, because the ideal cut-off points differ by problem size. The exception close to index value 1000 is due to the lack of the assumed perfect symmetry. The system is forced to arrive at a compromise using symmetric cut-off points.

Although the tabular approach reveals a performance gain, it comes with several disadvantages: it uses a huge amount of storage, it imposes upper limits to instance features (e.g. size), and it takes a long time to train (several days for the case above). This is mostly due to the lack of good generalization. The key observation here is that the running time of an algorithm typically varies smoothly as some instance feature changes smoothly. That makes generalization much easier compared to other domains.

Our second approach to learning makes use of linear architectures to represent the value function. The state $s = (n, d)$ in this case consists of the problem size n and the signed distance d of the index i from the midpoint of the array ($d = i - \lfloor n/2 \rfloor$). Using our knowledge about the shape of the value function, and after many trials, we found that the value function $Q(n, d, a)$ can be approximated by the following two parametric functions (one for each action/algorithm):

$$Q(n,d,a_D) = w_{1_D}\sqrt{1-\left(\frac{d}{n}\right)^2} + w_{2_D}\left(n - \frac{d^2}{n}\right),$$

$$Q(n,d,a_H) = w_{1_H}\sqrt{1-\left(\frac{2d}{n}\right)^2} + w_{2_H}\left(\frac{n}{4} - \frac{d^2}{n}\right),$$

where a_D, a_H are the actions of selecting DETERMINISTIC SELECT and HEAP SELECT respectively, and $w_{1_D}, w_{2_D}, w_{1_H}, w_{2_H}$ are the parameters (weights). Briefly, these functions represent a linear combination of a semi-ellipse and a parabola (for constant n). The amount of storage required in this case (see Section 4) is a 2×2 matrix ($\mathbf{A}^{(t)}$) and two 2×1 arrays ($\mathbf{q}^{(t)}$ and $\mathbf{w}^{(t)}$) for each equation. That gives a total of 16 real numbers which compares favorably with the ≈3735 numbers of the tabular case.

We trained the system on 2,400 randomly generated instances of different sizes distributed uniformly in the range $[20, 10000]$ with a schedule that starts with smaller sizes and moves toward larger sizes. The index was also varied uniformly within the available range for each size. We set the learning rate α to 1.0 for both actions to prevent use of wrong estimates and divergence of the value function. With $\alpha = 1.0$, only estimates of smaller sizes are used, since the resulting subproblems can only be smaller. As long as the training schedule is from smaller to larger sizes, it is guaranteed that these estimates will be fairly accurate, because training has been completed for smaller sizes. This idea is similar to the Grow-Support algorithm of Boyan and Moore (Boyan & Moore, 1995). Exploration was set to maximum ($\epsilon = 1$) so that both actions get approximately the same amount and distribution of data points. We used the least-squares approach, described in Section 4, to estimate the weights at each step during training.

The main advantage of the linear architecture is that the value function is defined for any state, even for states the system has not been trained on. Also, the learning time was less than an hour in this case. Overall, this second approach overcomes all the difficulties of the tabular approach with only a small degrade in performance (the best cut-offs cannot be estimated precisely due to the restricted form of approximation). Figure 4 shows performance results for fixed size ($n = 10000$) and Figure 5 results for

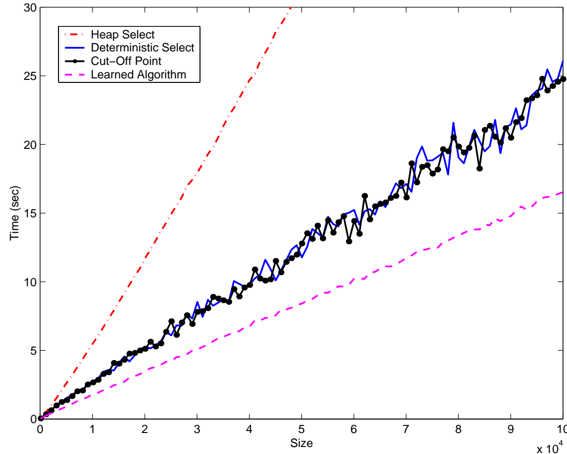

Figure 5. Order statistic selection (median, linear architecture).

fixed index ($d = 0$, the median) and size up to 100000. Note that the system was trained only on instances of size up to 10000. These initial results revealed that our approach to the algorithm selection problem is feasible and encouraged experimentation with other problems.

5.2 Sorting

The *sorting problem* is to rearrange an array of n (unordered) numbers in ascending order. This is probably the best known computational problem and there exist numerous sorting algorithms.

QUICKSORT (Cormen et al., 1990) is a recursive randomized sorting algorithm with $O(n^2)$ worst case running time and $O(n \log n)$ expected running time. It picks a partitioning element from the array at random and partitions the input in two parts such that all elements in the first part are less than or equal to the elements in the second part. Then, the two parts are sorted recursively. QUICKSORT is extremely efficient for large arrays.

INSERTIONSORT (Cormen et al., 1990) is a non–recursive algorithm with $O(n^2)$ worst case running time. It starts with the first element as the initial sorted list and iteratively inserts the other elements one–by–one at their correct position by shifting elements that are greater to the right. INSERTIONSORT is very efficient for small arrays and for inputs that are almost sorted.

A common approach is to run QUICKSORT for large sizes and switch to INSERTIONSORT when the size falls below some cut-off point. However, the optimal cut-off point may depend on several uncertain factors and it is unlikely fixed. Using our approach, however, it is possible to figure out the best cut-off point on the fly.

The state of the process consists of the size n of the input. Using our knowledge of the asymptotic running times, we approximate the value function (that is, the expected running time) by the following parametric functions (one for each algorithm):

$$Q(n, a) = w_1^{(a)} n^2 + w_2^{(a)} n \log_2 n + w_3^{(a)} n,$$

where a is either a_Q or a_I. The constant term is omitted, because $Q(0, a) = 0$ by definition. The weights for these linear architectures are estimated by the least-squares approach of Section 4.

We trained the system on 40 randomly generated instances only, 20 with size in $[1, 10]$, and 20 in $[10, 100]$, starting from smaller and moving toward larger sizes. We focus on this small range because the cut-off point lies somewhere in that range. The learning rate was set to 1.0 for reasons mentioned in the order statistic selection case. The learned weights were $w^{(Q)} = (0.006,\ -0.085,\ 5.969) \times 10^{-4}$ and $w^{(I)} = (0.142,\ -0.540,\ 3.539) \times 10^{-4}$. Figure 6 shows the learned value function along with actual running times for the individual algorithms. As expected, the value function for QUICKSORT is less than the actual running time of pure QUICKSORT, because of the ability to invoke INSERTIONSORT as needed. Notice that the cut-off point suggested by the learning algorithm is much lower than the point where the running time curves cross each other. This leads to an interesting insight: once a cut-off point is employed (say, the point where the running times cross), QUICKSORT becomes better overall, but INSERTIONSORT does not change. Thus, QUICKSORT can now be faster for instances right below the chosen cut-off point, where it was not faster before. That gives a new cut-off point and the same reasoning applies again and again, and the cut-off point moves lower and lower until it eventually converges. This is captured by the learning algorithm, but is difficult to work it out empirically offline.

Performance results are shown in Figure 7 for sizes up to 1000. The “cut-off point algorithm” is the one that uses the crossover point (size=47) of the running time curves. The learned algorithm, whose policy sets the cut-off point to size=35, performs around 15% better. The learned policy was precomputed beforehand to eliminate the overhead of evaluating the value function at each step.

6. Future Work and Conclusions

In this paper, we have ignored the distribution of the input data; all data come from the same uniform random distribution. Ideally, a learning system should be able to adapt rapidly to changes in the underlying distribution. To this end, it is required that (1) learning is continuously on, (2) some exploration is allowed, and (3) most recent data overshadow old data. We are currently experimenting with exponential windowing in our least-squares approach to ex-

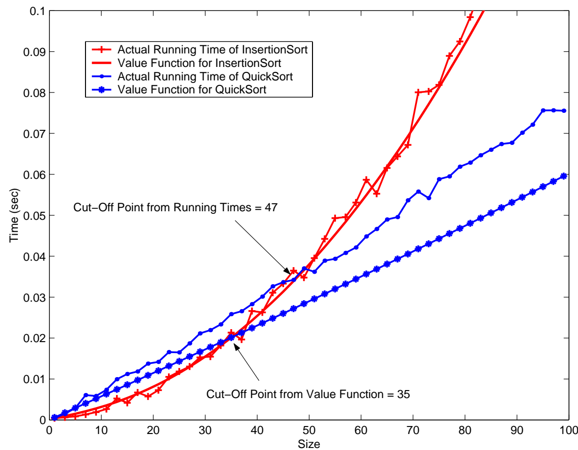

Figure 6. Value function, actual running time, and cut-off points.

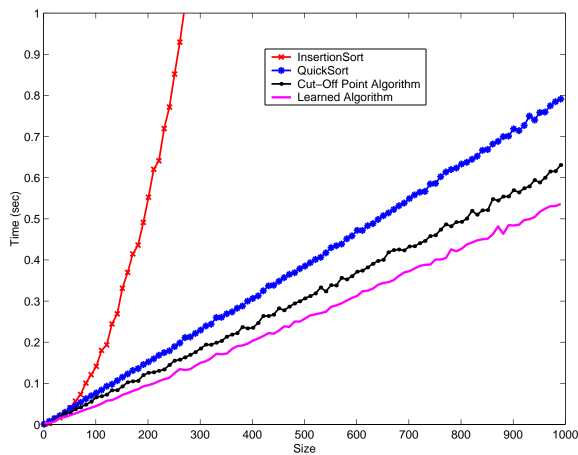

Figure 7. Running times for sorting.

ponentially discount old data. Allowing continuous exploration might lead to a cost penalty or to the discovery of a change. We have no clear solution to that problem, but in order to avoid unnecessary time penalties, we need more control over the algorithms. For example, we could terminate a selected (terminal) algorithm if its current running time exceeds significantly the estimate of the value function, and select another algorithm. We currently investigate these ideas on sorting. We plan to add more algorithms to the algorithm set and target for rapid online adaptation. We also plan to apply the proposed ideas to other problems, like convex hull and graph problems, where algorithm selection may induce significant savings.

The long-term goal and potential contribution of the work presented in this paper is twofold. First, from a computer science point of view, we envision an era where a computational problem is solved not by an isolated algorithm selected on the basis of its theoretical properties, but by an adaptive system that encapsulates the available repertoire of algorithms for that problem and selects them based mostly on their practical performance. We believe that such systems will be more efficient in applications that involve a wide and diverse range of problem instances. Second, from a machine learning point of view, the real-time constraint (learning is part of solving the problem) calls for learning algorithms that generalize and adapt rapidly while consuming minimum computational resources (especially time). The challenge for real-time learning becomes more and more important for the success of learning systems in real-world applications. The results presented here are the first steps along these directions and toward these goals.

Acknowledgments

The first author would like to thank the Lilian-Boudouri Foundation in Greece for financial support. The second author is supported in part by NSF-IRI-97-02576-CAREER.

References

Boyan, J. A. (1999). Least-squares temporal difference learning. *Machine Learning: Proceedings of the Sixteenth International Conference* (pp. 49–56). Morgan Kaufmann, San Francisco, CA.

Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. *Advances in Neural Information Processing Systems 7* (pp. 369–376). Cambridge, MA: The MIT Press.

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). *Introduction to algorithms*. Cambridge, MA: The MIT Press.

Fink, E. (1998). How to solve it automatically: Selection among problem-solving methods. *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems* (pp. 128–136). AAAI Press.

Lobjois, L., & Lemaître, M. (1998). Branch and bound algorithm selection by performance prediction. *Proceedings of the Fifteenth National Conference on Artificial Intelligence* (pp. 353–358). Menlo Park: AAAI Press.

Puterman, M. L. (1994). *Markov decision processes—discrete stochastic dynamic programming*. New York, NY: John Wiley & Sons, Inc.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. The MIT Press.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.