Selecting the Right Algorithm

Michail G. Lagoudakis*
Department of Computer Science
Duke University
Durham, NC 27708
mgl@cs.duke.edu

Michael L. Littman
Shannon Laboratory
AT&T Labs – Research
Florham Park, NJ 07932
mlittman@research.att.com

Ronald E. Parr
Department of Computer Science
Duke University
Durham, NC 27708
parr@cs.duke.edu

Computer scientists always strive to find better and faster algorithms for any computational problem. It is usually true that programmers and/or users come across a plethora of different algorithms when looking to solve a particular problem efficiently. Each one of these algorithms might offer different guarantees and properties, but it is unlikely that a single one of them is the best (fastest) in all possible cases. So, the question that the programmer/user typically faces is: *"Which algorithm should I select?"*

This question is largely due to the uncertainty in the input space, the inner workings of the algorithm (especially true for randomized algorithms), and the hardware characteristics. It's hard to know in advance what kind of inputs will be provided, how exactly the computation will proceed, or even how efficiently the underlying hardware will support the needs of the different algorithms. Sometimes, a careful study can reveal that committing to a particular algorithm is better than committing to any of the other algorithms, but is this the best we can do? What if uncertainty is explicitly taken into account and the right decision is made dynamically on an instance-by-instance basis?

To make the discussion more concrete, consider, for example, the problem of *sorting*. Why would you ever choose MergeSort or InsertionSort when you know that QuickSort is in general the fastest algorithm for sorting? That might be true to some extent, but it seems that if you allow for collaboration of these three "competitors", the outcome can be beneficial.

Think of this *algorithm selection problem* (Rice 1976) as a decision problem: *"Which algorithm should I run whenever a new instance is presented?"* The fact that two of these algorithms are recursive makes the problem even more interesting. Every time a recursive call is made, you can ask the same question: *"Which algorithm should I choose for the current subproblem?"*. Nothing really dictates that you have to use the same algorithm again and again throughout the recursion. How, then, can you optimize this sequence (or better, tree) of decisions? On what ground is each decision based?

Say that your decision is based only on the size of the input and that you have no other information about the input. You may assume that the elements in the input are not arranged in any particular order, but are uniformly distributed. Even if this is not the case, we can always force this by random scrambling. It seems that what we need then is a policy, that points out which algorithm to use for each possible input size.

It's not that hard to formulate this problem as a kind of Markov Decision Process (MDP). The state is the size of the input and the actions are the three algorithms. Choosing an action in a certain state results in a time cost that is paid for that choice, and a transition to one or two new states (recursive subproblems). It's easy to derive the transition model of the MDP. Action InsertionSort (I) takes you to the terminal state (sorted input) from any state with probability 1. Action MergeSort (M), if chosen in state s, takes you to two new states, $\lfloor s/2 \rfloor$ and $\lceil s/2 \rceil$, with probability 1. Finally, action QuickSort (Q), if taken in state s, will take you to a pair of states p, $s-p$, where p is 1 with probability 2/s and any value between 2 and $s-1$ with probability $1/s$ (Cormen, Leiserson, & Rivest 1990). Therefore, all transitions are to states of smaller size.

What is missing from our MDP is the cost function. However, we can estimate this function experimentally by running all algorithms on several inputs on the target machine and measuring the real execution time consumed for each transition.

By now, we have all the information we need to solve our problem using Dynamic Programming. Let's define $Opt(s)$ to be the minimum expected cost we can pay for sorting an input of size s. By definition, $Opt(1) = 0$ since an input of size 1 is already sorted. Now, suppose that we have $Opt(.)$ up to size $s-1$, then we can determine $Opt(s)$ recursively as follows:

$$Opt(s) = \min\{optQ(s), optI(s), optM(s)\}$$

where $optX(s)$, $X = \{I, Q, M\}$, is the total expected cost of taking action X in state s and following the optimal policy thereafter. $optX(s)$ can be determined as follows:

$$optI(s) = I(s)$$

$$optM(s) = M(s) + Opt(\lceil s/2 \rceil) + Opt(\lfloor s/2 \rfloor)$$

*Partially supported by the Lilian-Boudouri Foundation, Greece.

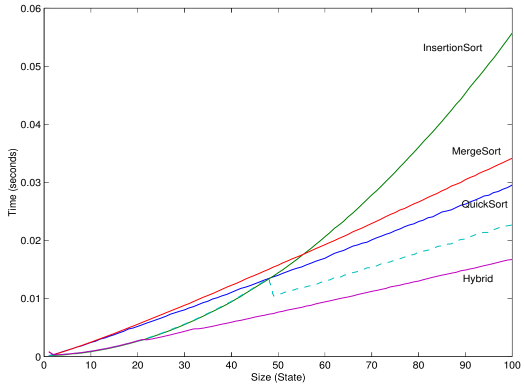

Figure 1: Average Performance of Sorting Algorithms on the Sun Sparc/Solaris Architecture

$$optQ(s) = Q(s) \quad + \quad \frac{1}{s}\sum_{p=1}^{s-1}\Big(Opt(p) + Opt(s-p)\Big)$$
$$+ \quad \frac{1}{s}\Big(Opt(1) + Opt(s-1)\Big)$$

where $X(s)$ is the immediate cost function we experimentally determined for each action X. Obviously, the optimal action for state (size) s would be the action that yields the minimum $Opt(s)$ in the equation above.

This algorithm was implemented and all calculations were performed up to size 100, since it is known that this is the critical area. Above that size QuickSort is definitely the optimal choice. The resulting optimal policies on two different machines are as follows:

Sparc	(Solaris)	Pentium	(Linux)
Size	Algorithm	Size	Algorithm
2 - 21	InsertionSort	2 - 17	InsertionSort
22 - 32	MergeSort	18 - 30	MergeSort
33 - ...	QuickSort	31 - ...	QuickSort

As expected, the resulting hybrid algorithm performs much better than any of the individual three algorithms. Figure 1 demonstrates savings of 43% over QuickSort, 50% over MergeSort and 69% over InsertionSort for inputs of size 100. The graph represents the average running time of each algorithm over 100 random instances (same for all algorithms) for sizes 1, 2, ..., 100.

Looking at the running time curves of InsertionSort and QuickSort and particularly at the point they cross each other ($\approx$ 48), you might be tempted to create yet another algorithm with the following policy:

Size	Algorithm
2 - 48	InsertionSort
49 - ...	QuickSort

The performance of this algorithm is shown with a dashed line and it is clearly not as good as the policy derived using the MDP. This is because interactions between the two algorithms are not taken into account (Lagoudakis & Littman 2000). Note also the discontinuity of the curve at the cut-off point. Such cut-off point algorithms are popular among the algorithmics community, but as shown with the example above a more formal approach can yield even better results.

There are some issues that arise in the context of recursive algorithm selection. We can assume random distribution for elements in the original input, but is this maintained for the subproblems during the recursion? Perhaps, subinstances become more and more sorted as we go, therefore the cost function is not valid anymore. If this is true, given that the distribution of the elements is hidden state, the result is a violation of the Markov property. Enhancing the state description with more information might reveal the hidden information, but it will make the derivation of a model practically impossible; a learning method will be necessary.

The resulting hybrid algorithm is somewhat fixed. What if some change in the underlying hardware or a different set of data results in a different cost function? Can we recompute the optimal policy dynamically? That is, can we continually update the individual cost function to be closer to reality, and solve the dynamic programming problem on the fly? This might be possible to some extent using a function approximator to represent the cost functions. The cost function is known to be linear for QuickSort and MergeSort, and somewhere between linear and quadratic in the average case for InsertionSort. The approximator should be able to adapt with only a few example points. The use of such function approximators has been investigated in (Lagoudakis & Littman 2000) for representing the value function in a model-free learning setting.

We focused on the problem of sorting in this paper. However, it is possible to extend these ideas to several other domains that enjoy portfolios of algorithms and recursive algorithms in particular. Promising results have been obtained on the problem of order statistic selection (Lagoudakis & Littman 2000) and on the problem of propositional satisfiability (Lagoudakis & Littman 2001).

This work demonstrates that learning and optimization methods can be effectively used to cope with uncertainty in computation. It is plausible that, in the future, problems will be solved by adaptive systems that encapsulate several algorithms and improve their performance using past experience.

References

Cormen, T. H.; Leiserson, C. E.; and Rivest, R. L. 1990. *Introduction to Algorithms*. Cambridge, MA: The MIT Press.

Lagoudakis, M. G., and Littman, M. L. 2000. Algorithm selection using reinforcement learning. In Langley, P., ed., *Proceedings of the Seventeenth International Conference on Machine Learning*, 511–518. Morgan Kaufmann, San Francisco, CA.

Lagoudakis, M. G., and Littman, M. L. 2001. Learning to select branching rules in the dpll procedure for satisfiability. In Kautz, H., and Selman, B., eds., *Electronic Notes in Discrete Mathematics (ENDM), Vol. 9, LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*. Elsevier Science.

Rice, J. R. 1976. The algorithm selection problem. *Advances in Computers* 15:65–118.