On Using Reinforcement Learning to Solve Sparse Linear Systems

Erik Kuefler and Tzu-Yi Chen*

Computer Science Department, Pomona College, Claremont CA 91711, USA
{kuefler,tzuyi}@cs.pomona.edu

Abstract. This paper describes how reinforcement learning can be used to select from a wide variety of preconditioned solvers for sparse linear systems. This approach provides a simple way to consider complex metrics of goodness, and makes it easy to evaluate a wide range of preconditioned solvers. A basic implementation recommends solvers that, when they converge, generally do so with no more than a 17% overhead in time over the best solver possible within the test framework. Potential refinements of, and extensions to, the system are discussed.

Keywords: iterative methods, preconditioners, reinforcement learning.

1 Introduction

When using an iterative method to solve a large, sparse, linear system $Ax = b$, applying the right preconditioner can mean the difference between computing x accurately in a reasonable amount of time, and never finding x at all. Unfortunately choosing a preconditioner that improves the speed and accuracy of the subsequently applied iterative method is rarely simple. Not only is the behavior of many preconditioners not well understood, but there are a wide variety to choose from (see, for example, the surveys in [1,2]). In addition, many preconditioners allow the user to set the values of one or more parameters, and certain combinations of preconditioners can be applied in concert. Finally, there are relatively few studies comparing different preconditioners, and the guidelines that are provided tend to be general rules-of-thumb.

To provide more useful problem-specific guidelines, recent work explores the use of machine learning techniques such as decision trees [3], neural networks [4], and support vector machines [5,6] for recommending preconditioned solvers. This line of research attempts to create a classifier that uses assorted structural and numerical features of a matrix in order to recommend a good preconditioned solver (with parameter settings when appropriate). At a minimum, these techniques recommend a solver that should be likely to converge to the solution vector. However, each paper also describes assorted extensions: [3] attempts to recommend a preconditioned solver that converges within some user-defined parameter of optimal, [5] attempts to give insight into why certain solvers fail,

* Corresponding author.

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 955–964, 2008.
© Springer-Verlag Berlin Heidelberg 2008

and [4] considers different use scenarios. In addition, [7] tries to predict the efficiency of a solver in terms of its time and memory usage, and [3] describes a general framework within which many machine learning approaches could be used. Other work explores statistics-based data mining techniques [8].

A drawback of the existing work is its dependence on *supervised* learning techniques. In other words, to train the classifier they need access to a large body of data consisting not only of matrix features, but also information on how different preconditioned solvers perform on each matrix. If the goal is predicting convergence, the database needs to keep track of whether a particular preconditioned solver with particular parameter settings converges for each matrix. However, if time to convergence is also of interest, the database must have consistent timing information. Furthermore, there must be an adequate number of test cases to allow for accurate training. These requirements may become problematic if such techniques are to be the basis of long term solutions.

An appealing alternative is reinforcement learning, which differs from previously applied machine learning techniques in several critical ways. First, it is *unsupervised* which means the training phase attempts to learn the best answers without being told what they are. This makes it easier to consider a large variety of preconditioned solvers since no large collection of data gathered by running examples is necessary for training the system. Second, it allows the user to define a continuous reward function which it then tries to maximize. This provides a natural way to introduce metrics of goodness that might, for example, depend on running time rather than just trying to predict convergence. Third, reinforcement learning can be used to actually solve linear systems rather than just recommending a solver.

After describing how reinforcement learning can be applied to the problem of choosing between preconditioned solvers, results of experiments using a basic implementation are discussed. Extensions and refinements which may improve the accuracy and utility of the implementation are also presented.

2 Using Reinforcement Learning

Reinforcement learning is a machine learning technique that tries to gather knowledge through undirected experimentation, rather than being trained on a specially-crafted body of existing knowledge [9]. This section describes how it can be applied to the problem of selecting a preconditioned iterative solver.

Applying reinforcement learning to a problem requires specifying a set of allowable actions, a reward (or cost) associated with each action, and a state representation. An agent then interacts with the environment by selecting an option from the allowable actions, and keeps track of the environment by maintaining an internal state. In response to the actions taken, the environment gives a numerical reward to the agent and may change in a way that the agent can observe by updating its state. As the agent moves within the environment, the agent attempts to assign a value to actions taken while in each state. This value is what the agent ultimately wishes to maximize, so computing an accurate

action-value function is the agent's most important goal. Note that the reward from taking an action in a state is different from its value: the former reflects the immediate benefit of taking that single action whereas the latter is a long-term estimate of the total rewards the agent will receive in the future as a result of taking that action.

The agent learns the action-value function through a training process consisting of some number of episodes. In each episode, the agent begins at some possible starting point. Without any prior experiences to guide it, the agent proceeds by performing random actions and observing the reward it receives after taking such actions. After performing many actions over several episodes, the agent eventually associates a value with every pair of states and actions. As training continues, these values are refined as the agent chooses actions unlike those it has taken previously. Eventually the agent will be able to predict the value of taking each action in any given state.

At the end of the training the agent has learned a function that gives the best action to take in any given state. When the trained system is given a matrix to solve, it selects actions according to this function until it reaches a solution.

2.1 Application to Solving Sparse Linear Systems

Reinforcement learning can be applied to the problem of solving sparse linear systems by breaking down the solve process into a series of actions, specifying the options within each action, and defining the allowable transitions between actions. Fig. 1 shows an example which emphasizes the flexibility of the framework. For example, the two actions labelled "scale" and "reorder," with transitions allowed in either direction between them, can capture the following (not unusual) sequence of actions: equilibrate the matrix, permute large entries to the diagonal, scale the matrix to give diagonal entries magnitude 1, apply a fill-reducing order. The implementation simply needs to allow all those matrix manipulations as options within the "scale" and "reorder" actions. Similarly, the single "apply iterative solver" step could include all the different iterative methods described in [10] as options. And every action can be made optional by including the possibility of doing nothing. Of course, increasing the flexibility in the initial specification is likely to increase the cost of training the system.

The state can be captured as a combination of where the agent is in the flowchart and assorted matrix features. These features should be cheap to compute and complete enough to represent the evolution of the matrix as it undergoes assorted actions. For example, features might include the matrix bandwidth or a matrix norm: the former is likely to change after reordering and the latter after scaling.

While the framework in Fig. 1 does allow for unnecessary redundant actions such as computing and applying the same fill-reducing heuristic twice, a well-chosen reward function will bias the system against such repetition. For example, a natural way to define the reward function is to use the time elapsed in computing each step. This not only allows the algorithm to see the immediate, short-term effects of the actions it plans to take, but also allows it to estimate

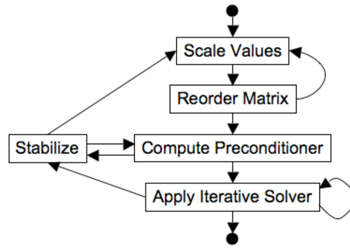

Fig. 1. One set of actions that could be used to describe a wide variety of solvers for sparse linear systems

the remaining time that will be required once that action is completed. In other words, the algorithm should be able to learn that taking a time-consuming action (e.g., computing a very accurate preconditioner) could be a good idea if it puts the matrix into a state that it knows to be very easy to solve. Notice that this means the framework gracefully allows for a direct solver (essentially a very accurate, but expensive to compute, preconditioner). In addition, if there are actions that result in failures from which there is no natural way to recover, those could be considered to result in essentially an infinite amount of time elapsing. If later a technique for recovery is developed, it can be incorporated ino the framework by adding to the flowchart.

Training the system consists of giving it a set of matrices to solve. Since the system must explore the space of possibilities and uses some randomness to do so, it should attempt to solve each matrix in the training set several times.

2.2 Implementation Details

The general framework for applying reinforcement learning to this problem is described above; important details that are specific to the implementation discussed in this paper are presented here.

First, the set of steps and allowable actions are restricted to those shown in Fig. 2. There are fewer actions than in Fig. 1, and the options within each action are restricted to the following:

- **Equilibrate:** The matrix can be initially equilibrated, or left alone.
- **Reorder:** The rows and columns of the matrix can be left unpermuted (natural), or one or the other could be reordered using a permutation computed using: MC64 [11,12], Reverse Cuthill-McKee [13], or COLAMD [14,15].
- **Precondition:** The preconditioner is restricted to the ILUTP_Mem [16] variant of incomplete LU, with one of 72 combinations of parameter settings: `lfil` between 0 and 5 inclusive, a `droptol` of 0, .001, .01, or .1, and a `pivtol` of 0, .1, or 1.
- **Solve:** The iterative solver is restricted to GMRES(50) [17] with a maximum of 500 iterations and a relative residual of $1e-8$.

The reinforcement learning framework allows for many more combinations of preconditioners than earlier studies which also restrict the solver to restarted

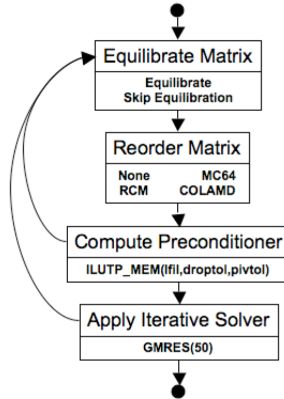

Fig. 2. Possible transitions between steps and their associated actions

GMRES and/or the preconditioner to a variant of ILU [4,5,6,7,18]. Observe, for example, that equilibration is now optional. Hence a total of 576 preconditioned solvers are described by the above framework; this is notably more than used to evaluate systems based on other machine learning techniques [3,4,5]. A system for automatically selecting from amongst so many options is particularly valuable given previous work that shows the difficulty of presenting information accurately comparing different preconditioned solvers across a range of metrics [19].

Note that because the state keeps track of where the program is in the flowchart, the system can restart the entire preconditioned solve if and only if the incomplete factorization breaks down or if GMRES fails to converge. As a result, the final system will be more robust since it can try different approaches if the first fails. While such step-based restrictions are not strictly necessary, incorporating domain knowledge by requiring the agent to perform computations in a logical order should reduce the training time and improve the accuracy of the trained system.

The state also keeps track of 32 structural and numerical features derived from the matrix itself. These are the same features as those used in [4], which are a subset of those used in [3,18]. Since each action changed the values of some of the features, this allowed the agent to observe the changes it made to the matrix during the computation and to react to those changes.

Finally, since the overall goal is minimizing the total time required to solve the matrices in the training set, the reward function used is the negative of the time required to complete that step. To bias the system against actions which are very fast but do not lead to a successful solve, the agent receives an additional reward (penalty) if GMRES fails to converge or if the ILU preconditioner cannot be computed. Without this safeguard, the agent might repeatedly take an action that cannot succeed and thus make no progress in learning the action-value function. The action-value function is initialized to 0, even though all true action values are negative. This is the "optimistic initial values" heuristic described in [9] that has the beneficial effect of encouraging exploration during early iterations of the

algorithm. Since the agent is effectively expecting a reward of 0 for each action, it will be continually "disappointed" with each action it takes after receiving a negative reward, and will thus be encouraged to experiment with a wide range of actions before eventually learning that they will all give negative rewards.

The high-level reinforcement learning algorithm was implemented in C++, with C and Fortran 77 used for the matrix operations. The code was compiled using g++, gcc, and g77 using the -O2 and -pthread compiler flags. The testing, training, and exhaustive solves were run on a pair of Mac Pro computers each running Ubuntu with 2 GB of RAM and four 2.66 GHz processors.

3 Experimental Results

The system described above was tested on a pool of 664 matrices selected from the University of Florida sparse matrix collection [20]. So that the results could be compared against the best results possible, all 576 preconditioned solvers allowed for by Fig. 2 were run on each matrix. However, due to time constraints, only 608 of the 664 matrices completed all 576 runs. Fig. 3 plots the number of matrices (out of 608) that converged for a given number of runs; note that each bar represents a decile. Every matrix converged for at least one setting, and 7 converged for all settings. Overall, 42% of the tested preconditioned solvers converged. For each matrix the fastest time taken to solve it was also saved so that the results using the trained system could be compared to it.

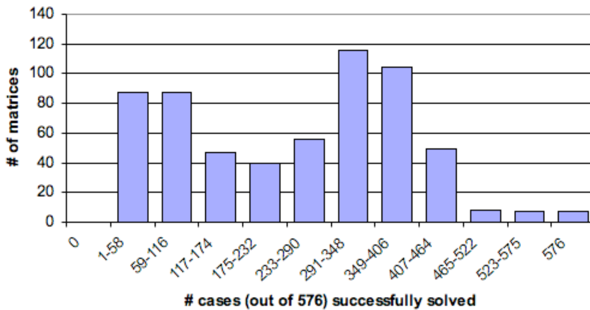

Fig. 3. Convergence results from testing all 576 possible preconditioned solvers on 608 of the matrices in the test suite. The y-axis gives the number of matrices which converged for some number of the solvers, the x-axis partitions 576 into deciles.

3.1 Methodology

The following protocol for training and testing was repeated 10 times.

The system was trained on 10% of the matrices, chosen at random, by solving each of those matrices 40 times. Since the framework restarts if the ILU factorization fails or GMRES does not converge, potentially many more than 40

attempts were made. As demonstrated in Fig. 3, every matrix can be solved by at least one solver so eventually repeated restarts should result in finding the solution.

After the training phase, the algorithm was tested on two sets of matrices. The first was equivalent to the training set; the second contained all 664 matrices. From each testing set the number of matrices successfully solved on the algorithm's first attempt (without a restart on failure) was calculated. Next, the ratio of the time taken to solve the matrix was divided by the fastest time possible within the framework described in Section 2.2. If the reinforcement learning algorithm did a good job of learning, this ratio should be close to 1. If it equals 1 then the algorithm learned to solve every matrix in the best possible way.

3.2 Results

Table 1 gives both the percentages of matrices that the system successfully solves on its first try and the time it took to solve them. These numbers are given both when the algorithm is tested on matrices in its training set and when it is tested on a more diverse set of matrices.

Table 1. Percent of systems successfully solved, and the median ratio of the time taken to solve those systems vs. the fastest solver possible, both when the testing and training sets are equivalent and when the testing set is larger and more diverse

	testing = training	testing = all matrices
percent solved	81.8%	56.4%
ratio of time	1.14	1.16

As expected, convergence results are best when the training and testing set are identical, with a success rate of 81.8%. When tested on the entire set of matrices, 56.4% of matrices were successfully solved (note that both of these percentages should go up if restarts are allowed). As was done for Fig. 3, Fig. 4 plots the number of matrices that were successfully solved in a given number of trials. Note that there were 10 trials overall and that, on average, a matrix should only be in the training set once. Comparing Fig. 4 to Fig. 3, observe that matrices were more likely to be solved in a greater percentage of cases, and that a larger number of cases converged overall (56% vs 42%). This indicates that the system has learned an action-value function that appropriately penalizes preconditioned solvers which cannot solve a system.

Since the time taken to solve each matrix must be compared to the optimal time (as computed through exhaustive search), the second row in Table 1 takes the ratio of solved time to best possible time and gives the median of those ratios. Note that this ratio could only be computed for the 608 matrices on which the full set of exhaustive runs was completed. While the results were slightly better when the training and testing sets were equivalent, overall half the matrices that were solved were done so with no more than 16% overhead over the fastest

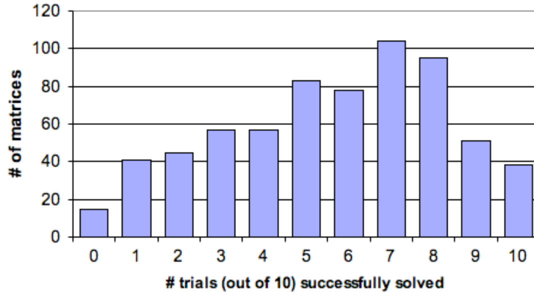

Fig. 4. The number of matrices which were correctly solved on the first try for a given number of trials (out of 10)

solution possible regardless of whether the matrix was in the testing set as well as the training set.

4 Discussion

This paper describes a framework for using reinforcement learning to solve sparse linear systems. This framework differs from that of previous sytems based on other machine learning techniques because it can easily factor running time into the recommendation, it makes it practical to consider a far larger number of potential preconditioned solvers, and it actually solves the system. In addition, the framework is extensible in the sense that it is simple to add new operations such as a novel iterative solver or a new choice of preconditioner.

An initial implementation that focussed on solving systems using ILU preconditioned GMRES is described. And while the convergence results presented in Section 3 are not as good as those in papers such as [4], the problem being solved here is more complex: rather than predicting if any of a set of preconstructed solvers would be likely to solve a particular matrix, this architecture creates its own solver as an arbitrary combination of lower level operations. Furthermore, the results are based on the system's first attempt at solving a problem — there was no possibility of a restart on failure since, without learning (which injects some randomness) in the final trained system, a restart without some matrix modification would result in the same failure. Note that either incorporating randomness (say by enabling learning) and allowing a restart after any kind of failure, or trying something more complex such as adding αI to A [21] upon a failure to compute the ILU preconditioner, should improve the convergence results. Of course, restarts would take time, so the ratio of time solved to best possible time would increase.

The fact that the code had trouble solving general-case matrices when the testing set is much more diverse than the training set suggests that the algorithm may not be generalizing sufficiently. This is a known issue in reinforcement learning (and all other machine learning techniques), and there are standard ways to attempt to improve this. Possibilities include a more sophisticated state

encoding (e.g., Kanerva Coding [22]), or reducing the set of matrix features used to define the state to those that are particularly meaningful (work on determining these features is currently underway). As with other machine learning techniques, there are also many opportunities to find better constants in the implementation. For the tested implementation values for parameters such as the number of training episodes, the learning rate, the eligibility trace decay, the size of tiles, and the number of tilings were chosen based on general principles and were experimented with only slightly.

An intruiging direction for future work is exploring alternative reward functions. Even within the current implementation a modified reward function that, say, punished failure more might improve the behavior of the trained system. But, in addition, the reward function could be modified to use any metric of goodness. For example, a function that depended on a combination of space and time usage could be used to build a recommendation system that would take into account both. And, in fact, one could imagine a personalized system for solving sparse linear systems that allows users to define a reward function which depends on the relative utilities they assign to a wide variety of resources.

Acknowledgements. The authors would like to thank Tom Dietterich for helpful discussions. This work was funded in part by the National Science Foundation under grant #CCF-0446604. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Benzi, M.: Preconditioning techniques for large linear systems: A survey. J. of Comp. Physics 182(2), 418–477 (2002)
2. Saad, Y., van der Vorst, H.A.: Iterative solution of linear systems in the 20th century. J. Comput. Appl. Math. 123(1-2), 1–33 (2000)
3. Bhowmick, S., Eijkhout, V., Freund, Y., Fuentes, E., Keyes, D.: Application of machine learning to the selection of sparse linear solvers. International Journal of High Performance Computing Applications (submitted, 2006)
4. Holloway, A.L., Chen, T.-Y.: Neural networks for predicting the behavior of preconditioned iterative solvers. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007. LNCS, vol. 4487, pp. 302–309. Springer, Heidelberg (2007)
5. Xu, S., Zhang, J.: Solvability prediction of sparse matrices with matrix structure-based preconditioners. In: Proc. Preconditioning 2005, Atlanta, Georgia (2005)
6. Xu, S., Zhang, J.: SVM classification for predicting sparse matrix solvability with parameterized matrix preconditioners. Technical Report 450-06, University of Kentucky (2006)
7. George, T., Sarin, V.: An approach recommender for preconditioned iterative solvers. In: Proc. Preconditioning 2007, Toulouse, France (2007)
8. Ramakrishnan, N., Ribbens, C.J.: Mining and visualizing recommendation spaces for elliptic PDEs with continuous attributes. ACM Trans. on Math. Softw. 26(2), 254–273 (2000)
9. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)

10. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., van der Vorst, H.: Templates for the solution of linear systems: Building blocks for iterative methods. SIAM, Philadelphia (1994)
11. Duff, I.S., Koster, J.: The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM J. Matrix Anal. Appl. 20(4), 889–901 (1999)
12. Duff, I.S., Koster, J.: On algorithms for permuting large entries to the diagonal of a sparse matrix. SIAM J. Matrix Anal. Appl. 22(4), 973–996 (2001)
13. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proc. of the 24th Natl. Conf. of the ACM, pp. 157–172 (1969)
14. Davis, T., Gilbert, J., Larimore, S., Ng, E.: Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. ACM Trans. on Math. Softw. 30(3), 377–380 (2004)
15. Davis, T., Gilbert, J., Larimore, S., Ng, E.: A column approximate minimum degree ordering algorithm. ACM Trans. on Math. Softw. 30(3), 353–376 (2004)
16. Chen, T.-Y.: ILUTP_Mem: A space-efficient incomplete LU preconditioner. In: Laganá, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA 2004. LNCS, vol. 3046, pp. 31–39. Springer, Heidelberg (2004)
17. Saad, Y., Schultz, M.H.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. 7(3), 856–869 (1986)
18. Xu, S., Zhang, J.: A data mining approach to matrix preconditioning problem. Technical Report 433-05, University of Kentucky (2005)
19. Lazzareschi, M., Chen, T.-Y.: Using performance profiles to evaluate preconditioners for iterative methods. In: Gavrilova, M.L., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganá, A., Mun, Y., Choo, H. (eds.) ICCSA 2006. LNCS, vol. 3982, pp. 1081–1089. Springer, Heidelberg (2006)
20. Davis, T.: University of Florida sparse matrix collection. NA Digest 92(42), October 16, 1994 and NA Digest 96(28) July 23, 1996, and NA Digest 97(23) June 7 (1997) http://www.cise.ufl.edu/research/sparse/matrices/
21. Manteuffel, T.A.: An incomplete factorization technique for positive definite linear systems. Mathematics of Computation 34, 473–497 (1980)
22. Kanerva, P.: Sparse Distributed Memory. MIT Press, Cambridge (1988)