Anytime Algorithm Development Tools

Joshua Grass and **Shlomo Zilberstein**
Computer Science Department
University of Massachusetts
Amherst, MA 01003 U.S.A.
{jgrass,shlomo}@cs.umass.edu

Abstract

Anytime algorithms are playing an increasingly important role in the construction of effective reasoning and planning systems. Early work on anytime algorithms concentrated on the construction of applications in such areas as medical diagnosis and mobile robot navigation. In this paper we describe a programming environment to support the development of such applications as well as larger applications in which several anytime algorithms are used. The widespread use of anytime algorithms depends largely on the availability of such programming tools for algorithm construction, performance measurement, composition of anytime algorithms, and monitoring of their execution. We present a prototype system that meets these needs. Created in lisp, this library of functions, graphical tools and monitoring modules will accelerate and simplify the process of programming with anytime algorithms.

1 Introduction

Anytime algorithms are algorithms that trade performance for time. As the amount of time is increased, an anytime algorithm improves the quality of the output. As computational systems become more complex and flexible, specific situations will demand more out of certain algorithms and less out of others. Since complete computation in many complex systems is not feasible, as well as not warranted, anytime algorithms provide a technique for allocating computational resources to the most useful algorithm. Currently, anytime algorithms are being used in a number of systems in such areas as diagnosis and repair, mobile robot navigations and decision under uncertainty [1, 2, 3, 5, 9, 13]. Anytime algorithms differ from normal algorithms in a number of ways:

1. **Quality measure** Instead of a binary notion of correctness, an anytime algorithm returns a result with a measure of it's quality. This may be a best guess, a group of possible answers, ranges, or other measurements.

2. **Predictability** Anytime algorithms also contain statistical information about the output quality given a certain amount of time and information about the data it receives. This can be used for meta-reasoning about computational resources to give an anytime algorithm.

3. **Interruptibility and Continuation** Anytime algorithms can be interrupted and return the partial results they have calculated up to that point or continued past the contract time they are given. This allows systems that use anytime algorithms to change the computation allocation to an anytime algorithm as it is executing.

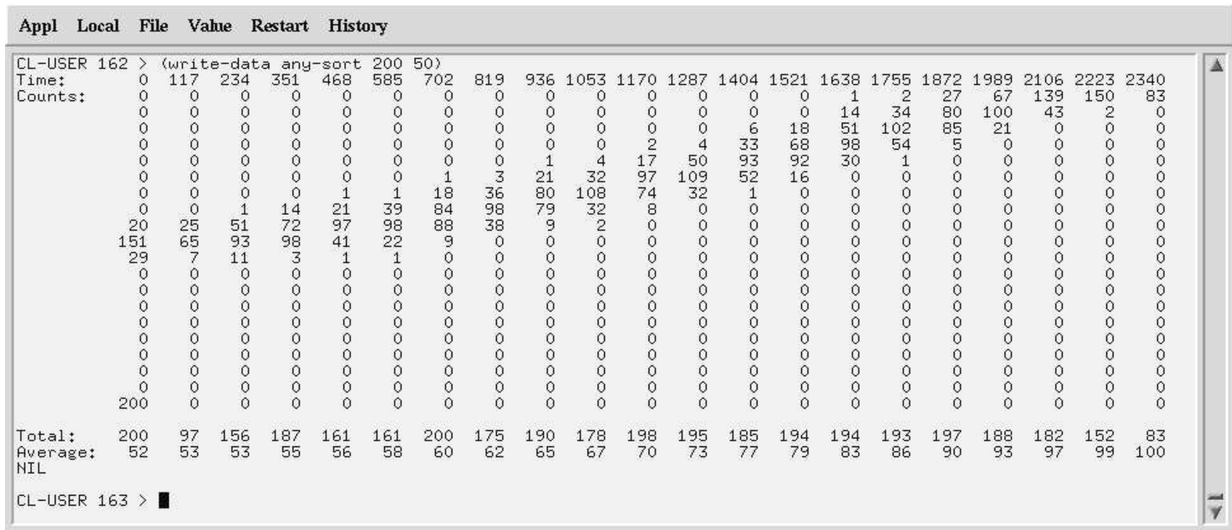
4. **Monotonicity** Anytime algorithms always improve the output quality of data they work on as they are given more time.

Anytime algorithms deal with data-structures in a slightly different way then traditional algorithms. In a traditional algorithm, internal information about where and how work is proceeding is maintained separately from the data, and thus interrupted functions cannot continue and often intermediate answers are lost. In an anytime algorithm, location information is maintained with the data and data manipulations are done in a way so that the data is still usable. Much like object-oriented data-structures maintain information about manipulators, anytime algorithm data-structures maintain information about the state of those manipulators as they work on the data. This information allows the system to make more intelligent decisions about the data and it's value. For example, a database may contain information about how sorted certain columns are and what needs to be done to finish the sorting along with an estimate of the time required. This type of information makes it possible to decide how to organize a search more effectively.

As the applicability of anytime algorithms grows, there is a growing need to develop programming methodologies and tools to support their unique characteristics. Current applications of anytime algorithms are based on incompatible representations of performance profiles and incompatible interface between the anytime components of the system. As a result, anytime algorithms are not reusable and each system must be developed from scratch.

In designing and developing an anytime programming environment, it is our hope that researchers in this field will be able to share useful libraries of anytime functions and easily compose those functions to create more flexible and powerful anytime systems. Additional tools could accelerate research in this field by allowing the user to visualize the performance of an anytime algorithm and analyze its properties. With the system presented in this paper, the user can easily develop additional anytime functions, examine and store their performance description, activate them and monitor their execution, and modify the existing library of programming tools.

Another goal of our system is to facilitate and standardize the anytime algorithm development cycle which is quite different from traditional algorithm development. This cycle begins with creating an iterative improvement algorithm in standard Common Lisp, then putting this algorithm through a tool that makes it into an anytime algorithm. The resulting algorithm can be activated with a certain time limit or it can be interrupted by the system. Once an anytime algorithm has been created, our system can automatically create a performance profile that relates input quality and computation time to the quality of the output. The combination of an anytime algorithm and its performance profile defines an anytime function. Our system also allows the composition of several elementary anytime functions into larger anytime systems. Finally, the user can select a monitoring strategy from a library (or define a new monitoring function) that would actually activate the anytime system and control the activation and interruption of the components.



```
Appl  Local  File  Value  Restart  History

CL-USER 162 > (write-data any-sort 200 50)
Time:        0  117  234  351  468  585  702  819  936 1053 1170 1287 1404 1521 1638 1755 1872 1989 2106 2223 2340
Counts:      0    0    0    0    0    0    0    0    0    0    0    0    0    0    1    2   27   67  139  150   83
             0    0    0    0    0    0    0    0    0    0    0    0    0    0   14   34   80  100   43    2    0
             0    0    0    0    0    0    0    0    0    0    0    0    6   18   51  102   85   21    0    0    0
             0    0    0    0    0    0    0    0    0    0    2    4   33   68   98   54    5    0    0    0    0
             0    0    0    0    0    0    0    0    1    4   17   50   93   92   30    1    0    0    0    0    0
             0    0    0    0    0    0    1    3   21   32   97  109   52   16    0    0    0    0    0    0    0
             0    0    0    0    1    1   18   36   80  108   74   32    1    0    0    0    0    0    0    0    0
             0    0    1   14   21   39   84   98   79   32    8    0    0    0    0    0    0    0    0    0    0
            20   25   51   72   97   98   88   38    9    2    0    0    0    0    0    0    0    0    0    0    0
           151   65   93   98   41   22    9    0    0    0    0    0    0    0    0    0    0    0    0    0    0
            29    7   11    3    1    1    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
             0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
             0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
             0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
             0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
             0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
             0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
             0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
             0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
           200    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0

Total:     200   97  156  187  161  161  200  175  190  178  198  195  185  194  194  193  197  188  182  152   83
Average:    52   53   53   55   56   58   60   62   65   67   70   73   77   79   83   86   90   93   97   99  100
NIL

CL-USER 163 >
```

Figure 1: The quality map of an anytime quick-sort function

The resulting system eliminates much of the redundant work involved in creating anytime systems and automates several related tasks from performance gathering to run-time monitoring. In addition to offering a basic set of programming tools, this programming environment offers a framework to study different methods of representing performance profiles, different techniques to combine anytime functions, and different monitoring schemes. The rest of this paper describes the components of our programming environment and its current capabilities. Section 2 describes the process of developing an elementary anytime algorithm. In Section 3, we show how composition of several modules is performed. Section 4 describes how monitoring of an anytime system is performed. We conclude with a summary of the benefits of the system and future work.

2 Developing elementary anytime algorithms

This section describes the process of developing a single anytime algorithm in our system. Our system supports and encourages the notion of modularity and re-usability of code. The difficulty with anytime algorithms is that we have to measure the time/quality tradeoff offered by each algorithm as well as test their correctness.

2.1 Extracting a single improvement step

The first step in creating an anytime function involves taking an iterative task, extracting a single quality improvement step and designing a data structure that can represent an intermediate state of the algorithm. A single quality improvement step must be a small computational task with respect to the work needed to solve the problem. This may be working on one element of an array, expanding a single node of a search graph, or running one pass with a filter. This function is passed to the anytime development system, which wraps time interruptible code around the algorithm and makes it into a lisp function that can run for a given amount of time or until the termination condition of the algorithm is met.

By recording the state of the process after each step, the anytime algorithm can work on data exactly as a normal iterative function would. While the basic processing step is kept the same, the order of the operation on the data may be modified to optimize partial performance.

The anytime algorithm is the function that actually does the computation on the structure that the system uses. Learning how to extract the basic quality improvement step from a given iterative algorithm is not hard, but does take a little practice. Two variables are needed by the anytime function. The first is a pointer to the data structure itself, this data structure will be destructively changed using a setf, so if the system needs the old version a copy must be made. The second variable is a location variable that keeps track of the anytime function's location in the data or returns true if the anytime function is complete. During each pass the anytime algorithm should do the smallest step possible, since at this point the function will not be interruptible. In the quick-sort case, the single improvement step consists of partitioning the array around a pivot in a given range of indices. A location variable is returned by the function after each pass (in Lisp, this location variable can be any data type that facilitates manipulating the data structure). It may be a list, vector, value or pointer. In the quick-sort instance we use a list of partition boundary pairs.

2.2 Support functions

Our system requires the user to supply two auxiliary functions. First, a sample generation function must be provided for creating random problem instances. These problem instances are used to automatically track the algorithm's performance and create its performance profile. The sample generation function should be designed to generate cases that would generally appear in the real-world application.

In addition, an output quality function must be provided that returns the *actual* output quality of the result structure. Determining a good measure of quality for the problem is important since it has a major effect on the accuracy and applicability of the performance profile. At the same time, a complex quality function may complicate the construction of the performance profile and may not be useful at run-time when quality of intermediate results must be determined within a short time. In our sorting example we use the percentage of elements that are less than their left neighbor.

2.3 Creating the performance profile

From an early stage, performance profiles played an important role in anytime computation. Early representations of performance profiles included a mapping from time allocation to *expected* output quality [1, 5]. This representation has been later extended by conditional performance profiles that include a mapping from input quality and run-time to a probability distribution of output quality [8, 10]. We use the latter type of performance profiles in our system.

Generation of the performance profile consists of several control structures that continually give the anytime algorithm small amounts of time and keep track of the new output quality. The system starts by estimating the average completion time of the algorithms by using a relatively small number of problem instances. This time is essential for determining the size and resolution of the table used to collect performance data. Then the system automatically generates a large number of problem instances and records the quality improvement of the algorithm as a function of time for each problem. This data, called the quality map of the algorithm, is used to construct the probability distribution of output quality for a given time. If input quality is variable, the system repeats the above process for a set of different initial input qualities so that the conditional performance profile of the algorithm can be constructed.

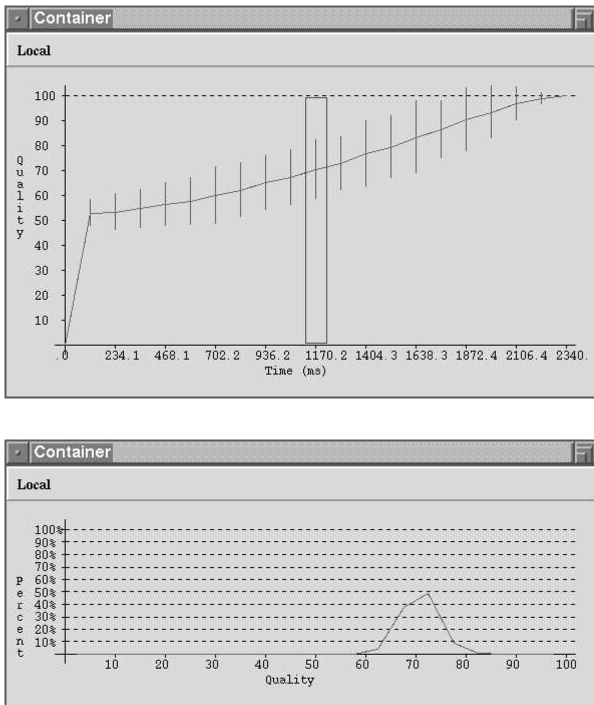


Figure 2: The performance profile of an anytime quick-sort algorithm (top) and the distribution of output quality at time 1170.2ms (bottom). Horizontal bars represent one standard deviation.

Figure 1 shows the 20 by 20 table representing the quality map of the quick-sort algorithm. Figure 2 shows how the system displays the resulting performance profile. The vertical axis represents output quality of the function on a scale of 0 to 100. This represents the quality measure defined by the quality function that the user supplies. In our sorting example, quality is determined by counting the number of entries that were less then their left neighbor. In a fully sorted list each number should be less then its left neighbor and this corresponds to quality of 100. The horizontal axis represents run-time measured in milliseconds. Other information in the window contains the average time for completion, the size of the list to be sorted (200 in this case) and the number of trials (200).

The discrete probability distribution of quality can be displayed graphically for any time period by the performance profile tool (see Figure 2). This discrete probability record does not cost much in terms of memory (since we are just storing 400 counts), but it allows a great deal of probabilistic information to be used in determining the output quality. As we will see latter, this performance representation allows us to make accurate calculations about the performance of a complex anytime function that includes several anytime functions as components.

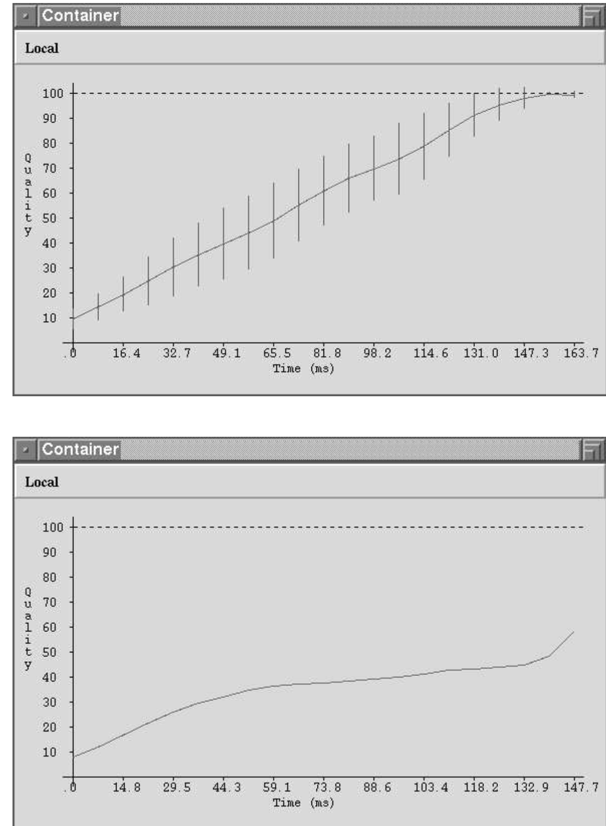


Figure 3: The performance profile of the anytime greedy selection algorithm with maximal input quality (top) and minimal input quality (bottom).

Quality maps can also be built for a number of different input qualities. The greedy-selection anytime algorithm, for example, will give different qualities of output depending on both the time allotted for it and the input quality (see Figure 3). The input quality in this case relates to the output quality of the anytime sort function. Thus some anytime functions have a conditional performance profile that captures the dependency on input quality.

3 Performance profiles of complex anytime systems

An important capability of our system is the automatic computation of performance profiles of complex anytime systems based on the performance profiles for the elementary components. We are currently implementing the compilation

techniques that were formulated in [10, 14]. These compilation techniques construct the best *contract*[1] algorithm for a complex anytime system. This contract algorithm can be made interruptible (if necessary) with only a small, constant penalty [8]. Our system is capable of compiling both linear chains of anytime algorithms and n-trees of anytime algorithms (see Figure 4).

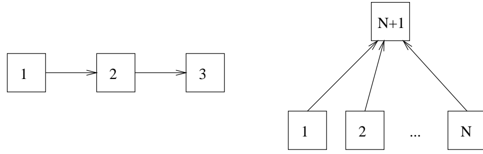


Figure 4: An anytime chain (left) and an anytime tree (right)

3.1 Compilation of an anytime chain

For example, consider the chain composition of two anytime algorithms. The first anytime algorithm manipulates data that the second anytime algorithm then uses to produces the final output. Combining anytime components in this manner allows us to build up more complex anytime systems from components that preprocess information. Chain compilation is the same as sequential function calls where the output from one function is the input of the next.

Fortunately, building chains of anytime functions only requires that we compose two components at a time. Local compilation has been proven to be optimal [10] so long as three constraints are true: The input function can be represented as a DAG, each anytime algorithm is monotonically increasing in quality with time, and the number of inputs for each module is bounded by a constant.

In order to construct the combined performance profile, the system first determines the maximum amount of time needed to generate output quality of 100 by both components. It is assumed that this time will be sufficient to generate an output quality of 100 for the combined function, and this time is used as the maximum time needed for the combined function. This time is then used in the creation of the performance profile of the combined function.

Once we have the maximum time from summing the maximum time of the components, we use a multiple resolution search to determine the optimal time allocation. A multiple resolution search allows the user to trade optimality for speed, by setting the number of division per time increment to search and the depth of search. The multiple resolution search divides the time window into n divisions, picks the best time(see below) and creates a new time window that is one time division below and above the best time. The compiler continues narrowing the time window until it has reached the depth specified.

In figure 5 we can see the compiler looking at one group of time points and then using the best time point to narrow the search to the next time point. Resolution is $time/divisions^{depth}$ and compile time is $O(divisions*depth)$. Since a small number of divisions may make the compiler miss an optimal time allocation the user must experiment with different time divisions and depths to determine the best compilation.

[1] A contract algorithm offers a tradeoff between computation time and output quality but the amount of time available form computation must be determined when the algorithm is activated [10].

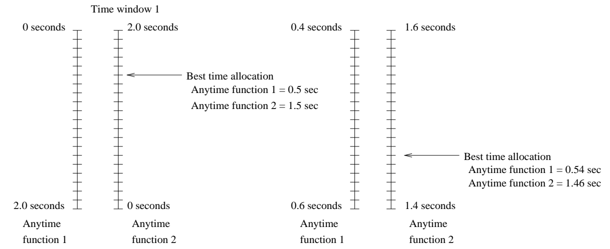


Figure 5: Multiple resolution search with divisions=20, and depth=2

Once a *specific* time allocation for t_N(the first anytime function) and t_M(the second) have been picked by the multiple resolution search, we can use the corresponding output quality distribution of the first function and the conditional performance profile of the second to determine the output distribution of the combined function. Figure 6 shows graphically how t_N and t_M are used to index the conditional performance profiles of the two functions. The best t_N and t_M are selected based on the expected output quality of the combined function. The expected output quality is determined by the following equation:

$$E\{Q_O\} = \sum_{q_j}(\sum_{q_i} Q_N(t_N)[q_i]Q_M(q_i, t_M)[q_j])q_j$$

Where:

q_i	is the output quality of the first function
q_j	is the output quality of the second function
Q_O	is the overall output quality
Q_N	is the performance profile of the first function
Q_M	is the performance profile of the second function
t_N	is the time allocation to the first function
t_M	is the time allocation to the second function

For any total time allocation t, the best time allocation to the components (t_N to the first function and t_M to the second) is recorded as part of the combined performance profile for monitoring purposes. The score is used to determine if this is the best time allocation for the current time window.

3.2 An example of an anytime chain

Using the system, we combined the greedy selection algorithm and the quick-sort anytime. The greedy anytime function was tested with initial qualities of 50, 70, 90, 95, 97 and 99. We focused more on the higher input qualities since this is where we need more accurate data. We used a minimum input quality of 50 since a randomized list has an input quality of 50 by our rating function. Figure 3 shows the resulting performance profile for maximal and minimal input qualities. Notice in Figure 7 that with low input quality, the output quality distribution is much broader then with high quality, making output quality more predictable as input quality increases.

The quick-sort and the greedy selection algorithms can be combined to solve the activity-selection problem in process management. Given a set of jobs characterized by their start and an end times, the activity selection algorithm guarantees to maximize the number of jobs completed in a specific time period. The list of possible jobs is first sorted by end times and then the selection algorithm runs through the sorted list selecting jobs that leave the least amount of free space, namely, selecting the first job with a start time after the end time of the current job. Both these algorithms demonstrate markedly different performance profiles and how

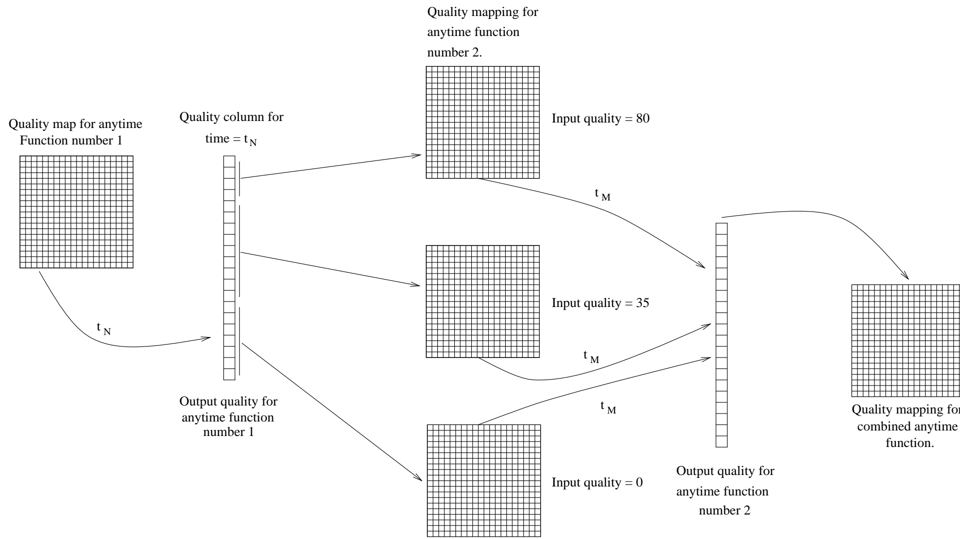


Figure 6: Determining the output quality distribution of a combined function based on the performance profiles of its components

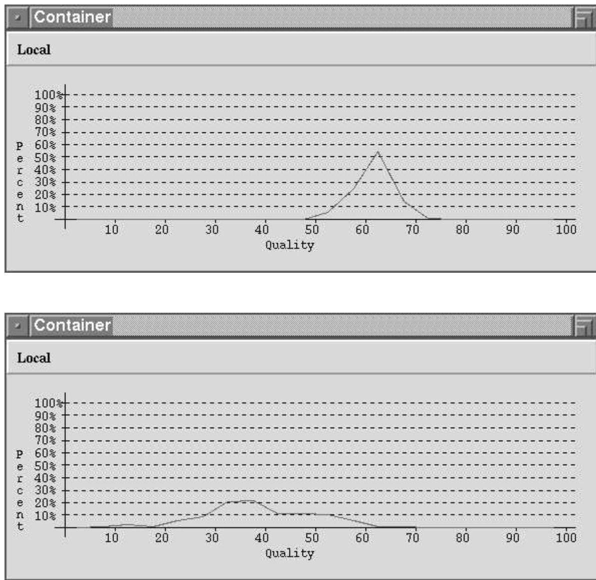


Figure 7: The distribution of output quality of the greedy selection algorithm with maximal input quality (top) and minimal input quality (bottom) for the same time allocation.

two functions can be combined to create a more complicated and useful anytime function.

After calculating the best time allocation for both of the anytime functions and the probabilistic output quality, the system displays the combined performance profile (see Figure 8).

The process of combining anytime functions can be applied to more complex structures. We have implemented the *local compilation* technique introduced in [10]. Figure 9, for example, shows the composition of three test functions. The local compilation technique works by first combining the performance profiles of function two and function three and then combining the result with the performance profile of function one. Since the performance profiles of complex anytime functions contain all of the same information that an elementary function does, complex anytime functions may be manipulated and used in exactly the same manner as elementary functions.

3.3 Compile N-tree Anytime Functions

Chain compilation is actually just a subset of n-tree compilation. Where the root of a chain anytime function has a performance profile indexed on one input quality, an N-tree root needs N input qualities to index the performance profile. And the multiple resolution search must go through all the possible time allocations for the root and each of its children. Figure 10 shows the multiple resolution search for one time window.

The scoring function also becomes slightly more complicated with the compilation of N-trees.

$$E\{Q_O\} = \sum_{q_j}\Big(\sum_{q_{i_1},\cdots,q_{i_L}} Q_N(t_{N_1})[q_{i_1}]\cdots Q_N(t_{N_L})[q_{i_L}]\; Q_M(q_{i_1},\cdots,q_{i_L},t_M)[q_j]\Big)q_j$$

Where:

$q_{i_1}\cdots q_{i_L}$	are the output qualities of the leaf functions
q_j	is the output quality of the second function
Q_O	is the overall output quality
Q_N	is the performance profile of the first function
Q_M	is the performance profile of the second function
$t_{N_1}\cdots t_{N_L}$	are the time allocations to the leaf functions
t_M	is the time allocation to the second function

Just as with the chain function the compilation of N-trees requires the scoring of all possible time allocations to the children and the parent. The compiler then selects the best time allocation, and creates a new time window around that time allocation.

3.4 An N-tree Example

In this example we create an anytime function called merge. Merge takes two sorted lists and outputs the merged results, it does this by calling two anytime sort functions on two lists and passing the results to the anytime merge function. The compiler determines the best time allocations by using the multiple resolution search technique described above. In

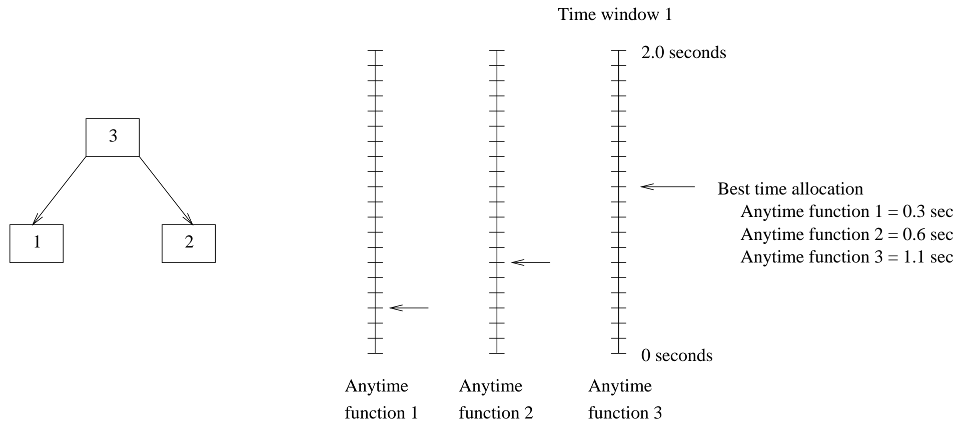


Figure 10: The time allocation of a binary tree anytime compilation

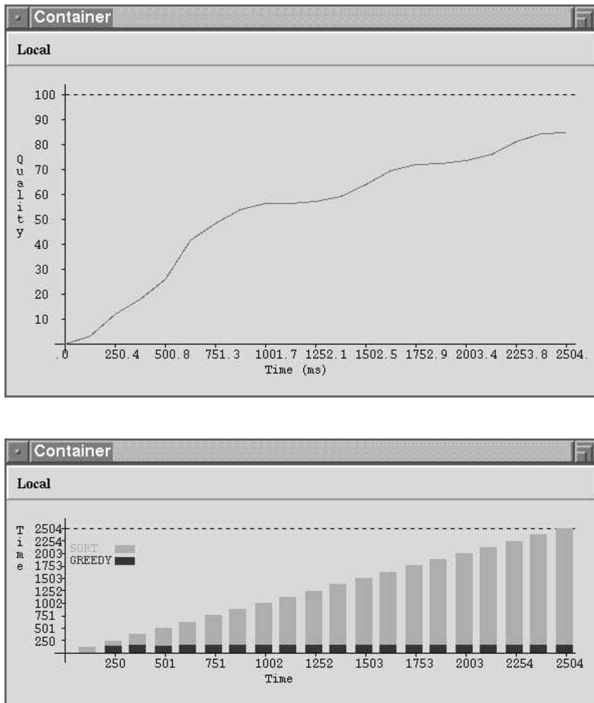


Figure 8: The combined performance profile of the quick-sort and greedy selection algorithms (top). The optimal time allocation (bottom) to quick-sort (light-gray) and to greedy selection (dark-gray).

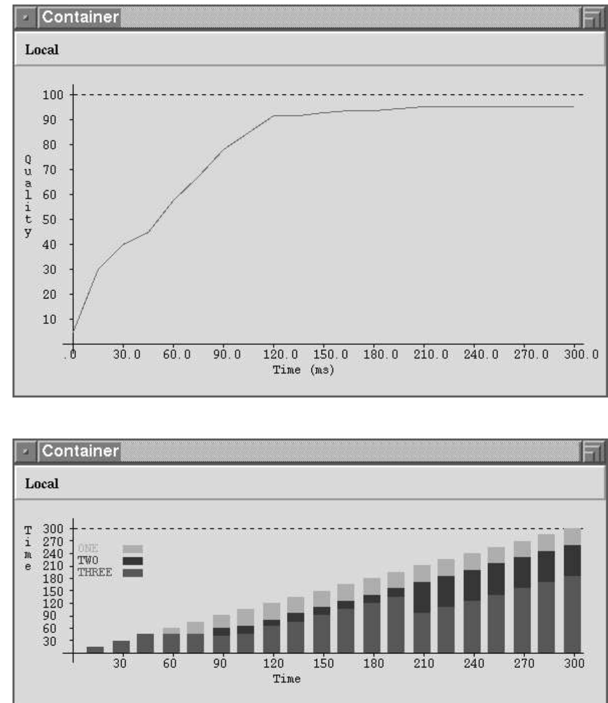


Figure 9: The combined performance profile of a three component anytime function (top). The optimal time allocation (bottom) for function one (light gray), function two (dark gray) and function three (medium gray).

figure 11 we can see the results of the compilation, the system displays both the performance profile and the optimal time allocation.

Just as with compiling anytime chains, anytime n-trees can be combined with other trees and chains to build more complex anytime functions. Since anytime n-tree structures have the same performance profiles, they can be manipulated in the exact same manner as an elementary anytime function.

4 Monitoring

Monitoring of execution is an important component of an anytime system. Two monitoring strategies have been developed for the system. The first is a fixed-contract monitoring scheme and the second is an adaptive monitoring scheme.

Both monitoring strategies use a time-dependent utility function (TDUF) that the user must supply. This function describes the value of approximate results produced by the anytime system in various situations. Our system passes the TDUF two arguments: the time and the quality of the results. Using this information the TDUF returns a utility based on the current environment, and the task trying to be accomplished. For example, the time allocated to path planning may not be important during normal operation, but when moving objects are nearby, path-planning time may be needed for reactive obstacle avoidance.

The monitor is designed to control the activation and interruption of the anytime algorithms so as to maximize overall utility. Figure 12 shows how the system combines the TDUF

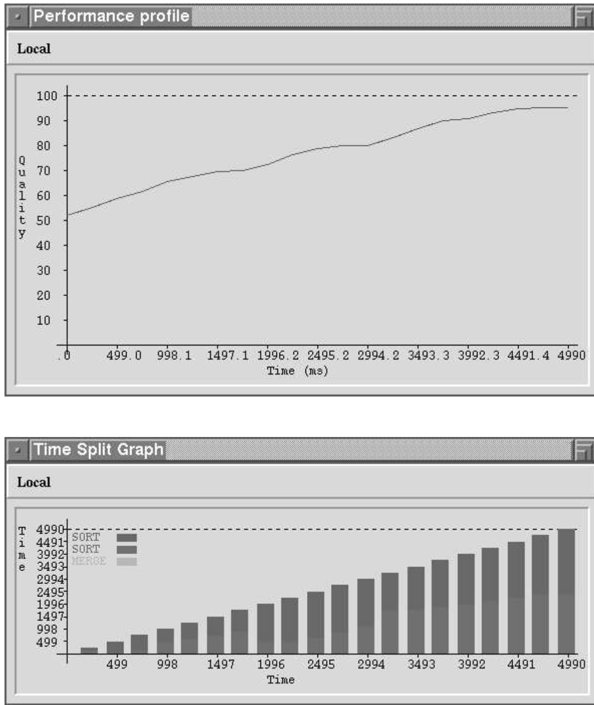


Figure 11: The combined performance profile for the binary-tree anytime merge function(top). The optimal time allocations for each of the three sub-functions(bottoms).

with the performance profile to generate a time vs. utility function, from which it can determine the best time allocation.

The use of TDUFs to define the objective of the monitor allows us to develop generic monitoring schemes that may be useful in many applications. In addition, the user need not know how to access the performance profiles or develop resource allocation algorithms (unless the "standard" available monitors are not suitable for the particular application).

The contract monitoring scheme uses the TDUF once to determine the optimal time allocation to the anytime function. Once the total time allocation is determined, the anytime function is run for that amount of time and the TDUF is not checked again. The monitor receives no more information

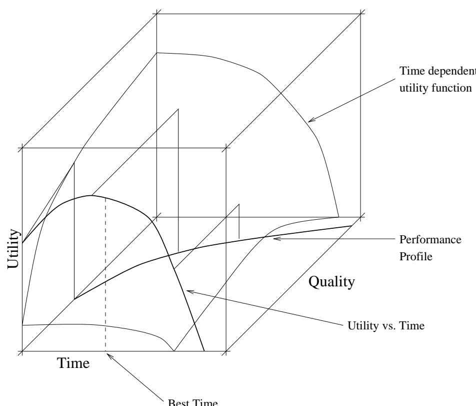
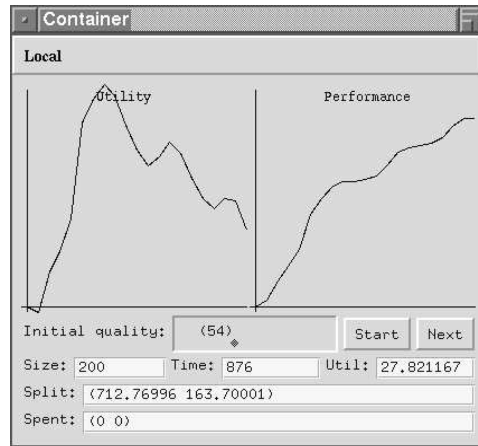


Figure 12: Using a TDUF and a performance profile to determine the best amount of time to allocate to an anytime function

from the anytime function.



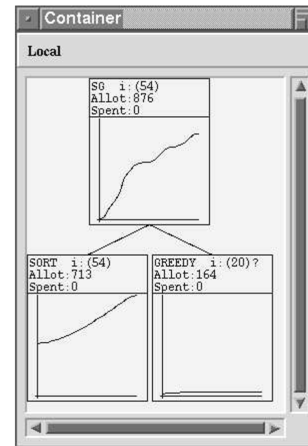


Figure 13: The graphical display for the adaptive monitoring scheme.

Figure 13 shows the adaptive monitoring scheme. Initially, adaptive monitoring behaves much like the fixed-contract scheme, using the TDUF to select the time allocation that will maximize utility. Instead of allocating time to the complex anytime function, the adaptive monitor uses the subpart time allocations in the complex anytime record to assign time to each sub-part. Then the adaptive monitor executes each sub-component and re-evaluates the TDUF, at each stage with more information about how the anytime algorithm is progressing and with current information about the environment. This allows the monitor to adapt to environment changes or unexpected data quality by reallocating time to the rest of the sub-components.

The monitor displays information about the intermediate quality of the data, the performance profiles of the subfunctions and the expected utility based on the time spent so far, the state of the world and the state of the data.

Once more monitoring strategies are developed, the user will be able to select a monitor based on the characteristics of the domain of operation and the nature of the time-dependent utility function. Several monitoring strategies that are currently under development have been analyzed in [10]. They offer provably optimal solutions to several general classes of domains. Examples include:

1. Scheduling contract algorithms in an environment with predictable utility change.

2. Adaptive modification of contract time based on monitoring the actual change in the environment and actual quality of results produced by the system.

3. Scheduling interruptible algorithms based on the marginal value of computation criterion.

5 Conclusions

We have described a prototype system for programming with anytime algorithms. Working on this project has given us an opportunity to study anytime algorithms as a new class of functions instead of just dealing with the development of a single anytime algorithm. Although many traditional programming techniques *can* produce useful anytime algorithms, programming with anytime algorithms is different from traditional programming. Nevertheless, we found that the ideas of modularity and re-usability are largely applicable in anytime system development, but anytime systems impose new requirements on the programmer.

We are still working at expanding and improving the anytime programming environment. Current effort is aimed at increasing the scope of compilation, implementing additional monitoring strategies, and improving the capabilities of the interactive graphical tools.

Acknowledgements

This work was partially supported by the University of Massachusetts under a Faculty Research Grant and by the National Science Foundation under grant IRI-9409827.

References

[1] M. Boddy and T. L. Dean. Solving time-dependent planning problems. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 979–984, Detroit, Michigan, 1989.

[2] T. L. Dean and M. Boddy. An analysis of time-dependent planning. *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 49–54, Minneapolis, Minnesota, 1988.

[3] C. Elkan. Incremental, approximate planning: Abductive default reasoning. *Proceedings of the AAAI Spring Symposium on Planning in Uncertain Environments*, Palo Alto, California, 1990.

[4] B. Hayes-Roth, R. Washington, R. Hewett, M. Hewett and A. Siever. Intelligent monitoring and control. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 243–249, Detroit, Michigan, 1989.

[5] E. J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. *Proceedings of the 1987 Workshop on Uncertainty in Artificial Intelligence*, Seattle, Washington, 1987.

[6] E. J. Horvitz and J. S. Breese. *Ideal partition of resources for metareasoning.* Technical Report KSL-90-26, Stanford Knowledge Systems Laboratory, Stanford, California, 1990.

[7] K. J. Lin, S. Natarajan, J. W. S. Liu and T. Krauskopf. Concord: A system of imprecise computations. *Proceedings of COMPSAC '87*, pp. 75–81, Tokyo, Japan, 1987.

[8] S. J. Russell and S. Zilberstein. Composing Real-Time Systems. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pp. 212–217, Sydney, Australia, 1991.

[9] K. P. Smith and J. W. S. Liu. Monotonically improving approximate answers to relational algebra queries. *COMPSAC-89*, Orlando, Florida, 1989.

[10] S. Zilberstein. Operational Rationality through Compilation of Anytime Algorithms. Ph.D. Dissertation, (also Technical Report No. CSD-93-743), Computer Science Division, University of California, Berkeley, 1993.

[11] S. Zilberstein Optimizing Decision Quality with Contract Algorithms. *Proceedings of the fourteenth International Joint Conference on AI*, pp. 1576-1582, Montreal, Canada, 1995.

[12] S. Zilberstein and S. J. Russell. Efficient resource-bounded reasoning in AT-RALPH. *Proceedings of the First International Conference on AI Planning Systems*, pp. 260–266, College Park, Maryland, 1992.

[13] S. Zilberstein and S. J. Russell. Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1402–1407, Chambery, France, 1993.

[14] S. Zilberstein and S. J. Russell. Optimal Composition of Real-Time Systems. *Artificial Intelligence*, forthcoming, 1995.