ELSEVIER

Artificial Intelligence 82 (1996) 181–213

Artificial Intelligence

Optimal composition of real-time systems

Shlomo Zilberstein *, Stuart Russell

Computer Science Division, University of California, Berkeley, CA 94720, USA

Received November 1993; revised October 1994

Abstract

Real-time systems are designed for environments in which the utility of actions is strongly time-dependent. Recent work by Dean, Horvitz and others has shown that *anytime algorithms* are a useful tool for real-time system design, since they allow computation time to be traded for decision quality. In order to construct complex systems, however, we need to be able to compose larger systems from smaller, reusable anytime modules. This paper addresses two basic problems associated with composition: how to ensure the *interruptibility* of the composed system; and how to allocate computation time optimally among the components. The first problem is solved by a simple and general construction that incurs only a small, constant penalty. The second is solved by an off-line compilation process. We show that the general compilation problem is NP-complete. However, efficient local compilation techniques, working on a single program structure at a time, yield globally optimal allocations for a large class of programs. We illustrate these results with two simple applications.

1. Introduction

This paper describes work on a fundamental problem in computer science and artificial intelligence, namely the construction of systems that can operate robustly in a variety of real-time environments. A real-time environment can be characterized by a *time-dependent* utility function. In almost all cases, the deliberation required to select optimal actions will degrade the system's overall utility. It is by now well-understood that a successful system must trade off decision quality for deliberation cost [2,4,15,21,29,31,32].

* Present address: Computer Science Department, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA 01003, USA.

SSDI 0004-3702(94)00074-3

The problem of deliberation cost has been widely discussed in artificial intelligence, economics and philosophy. In artificial intelligence in particular, researchers have proposed a number of meta-level architectures to control the cost of base-level reasoning [5, 6, 8, 12, 15, 28]. One promising approach is to use *anytime* [7] or *flexible* [14] algorithms, which allow the execution time to be specified, either as a parameter or by an interrupt, and exhibit a time/quality tradeoff defined by a *performance profile*. They provide a simple means by which a system can control its deliberation without significant overhead.

Soon after the introduction of anytime algorithms, it became apparent that their composition presents a vital, nontrivial problem [7]. This paper reports the first results on the composition problem showing that real-time systems can be modularly composed of anytime algorithms. Moreover, the meta-level scheduling problem is solved in polynomial time to yield optimal (near-optimal) performance for any tree (directed acyclic graph) structured program. These results extend the advantages of anytime algorithms to the design of complex real-time systems with many components.

In standard algorithms, the fixed quality of the output allows for composition to be implemented by a simple call-return mechanism. When algorithms have resource allocation as a degree of freedom, and can be interrupted at any time, the situation becomes more complex. Consider the following simple example: a real-time medical expert system containing a diagnosis component which passes its results to a treatment-planning component. The following issues arise:

(1) How can the individual components be designed as anytime algorithms?
(2) How can their performance be described as a function of time and the nature of the inputs?
(3) How does the output quality of the treatment component depend on the accuracy of the diagnosis it receives?
(4) What sort of programming language constructs are needed to specify how the system is built from its components?
(5) For any given amount of time, how should that time be allocated to each of the components?
(6) What if the condition of the patient suddenly requires intervention while the diagnosis component is still running and no treatment has been considered?
(7) How should the execution of the composite system be managed so as to optimize overall utility?

In other publications, particularly [34], we address these issues in some depth. Here, we focus on item (5), which we call the *compilation* problem. Given a system composed of anytime algorithms, compilation determines off-line the optimal allocation of time to the components for any given total allocation. The crucial meta-level knowledge for solving this problem is kept in the *anytime library* in the form of *conditional performance profiles*. These profiles characterize the performance of each elementary anytime algorithm as a function of run-time and input quality. In Section 2, we define the basic properties of anytime algorithms. An important distinction is made between *contract* algorithms, which require the determination of the total run-time when activated, and *interruptible* algorithms, whose total run-time is unknown in advance. The reduction theorem shows how to construct an interruptible algorithm once a contract algorithm is

compiled. In Section 3, we define the compilation problem and present a simple example of compilation. Then, in Section 4, we analyze in detail the compilation of functional composition. While the general compilation problem is shown to be NP-complete in the strong sense, local compilation techniques, whose complexity is linear in the size of the program, are shown to be both efficient and optimal for a large class of programs. In addition, a number of efficient approximation algorithms are given for the general case. Finally, Section 5 summarizes the benefits of compilation and outlines the direction for further work in this field.

2. Anytime algorithms

The term "anytime algorithm" was coined by Dean in the late 1980s in the context of his work on time-dependent planning. Anytime algorithms expand upon the traditional view of a computational procedure as they offer to fulfill an entire spectrum of input–output specifications, over the full range of run-times, rather than just a single specification. A standard algorithm is an implementation of a mapping from a set of inputs into a set of outputs. For each input that specifies a problem instance there is a particular element in the output set that is considered the *correct* solution to be generated by the algorithm. An anytime algorithm is an implementation of a mapping from a set of inputs and time allocation into a set of outputs. For each input there is a corresponding set of possible outputs, each of which is associated with a particular time allocation and some measure of its quality. The advantage of this generalization is that computation can be interrupted at any time and still produce results of a certain quality, hence the name "anytime algorithm".

2.1. Measuring the quality of results

In the context of anytime algorithms, a *quality measure* is typically a function from the output of the algorithm to the [0,1] interval. It may or may not be related to the utility function of the system that incorporates the algorithm; but it should measure some aspect of the algorithm's output that improves over time, at least on average. The following three metrics have proved useful in anytime algorithm construction:

(1) *Certainty*. This metric reflects the degree of certainty that a result is correct. The degree of certainty can be expressed using probabilities, fuzzy set membership, or any other method of expressing uncertainty. For example, consider an anytime diagnosis algorithm that is based on combining more and more evidence as computation time increases. The certainty that the diagnosis is correct increases as a function of run-time, but there remains a possibility that the correct result is entirely different from the result generated by the algorithm.

(2) *Accuracy*. This metric reflects the degree of accuracy in the value returned by the algorithm, typically through a bound on the difference from the exact solution. For example, if a Taylor series is being used to approximate a certain function, then the error bound (given by Lagrange's remainder formula) decreases with the iteration number. This error bound determines the quality of the results.

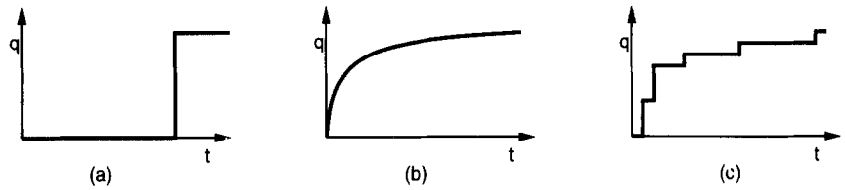

Fig. 1. Typical performance profiles: (a) Standard algorithm. (b) Idealized anytime algorithm. (c) Actual anytime algorithm.

(3) *Specificity*. This metric reflects the level of detail of the result. In this case, the anytime algorithm always produces *correct* results, but the level of *detail* is increased over time. For example, consider a hierarchical diagnosis algorithm that pinpoints a subassembly as the source of the fault. Over time, this can be refined all the way down to primitive components, but at any point the output is correct, even if not fully specific.

Notice that accuracy, a standard measure in numerical domains, can be mapped onto specificity, which is more commonly used in symbolic domains. An inaccurate numerical solution is very specific but incorrect, and could be mapped to an equally useful, correct statement that the solution lies within a certain interval. Anytime algorithms can also have multidimensional quality measures. For example, PAC algorithms for inductive learning are characterized by an uncertainty measure, δ, and a precision measure, ϵ.

2.2. *Performance profiles*

The *performance profile* of an algorithm characterizes the quality of its output as a function of computation time. All algorithms—whether standard or anytime—have a performance profile. Fig. 1 shows typical performance profiles for standard algorithms (a) and idealized anytime algorithms (b). The performance profile of the standard algorithm shows that no results are available until its termination at which point the exact result is returned. The idealized anytime algorithm provides output whose quality improves gradually over time. In practice, the improvement in quality of an anytime algorithm may look more like the profile shown in Fig. 1(c).

Strictly speaking, such profiles are defined only for a particular input, and only for deterministic algorithms. We will also need to describe the output quality for a population of inputs, and for a set of runs of a randomized algorithm. Further refinements are needed to describe how the performance depends on various aspects of the input such as quality and size. We also need ways to *acquire* and *represent* profiles. These issues are dealt with in the following subsections.

2.2.1. *Categories of performance profiles*

Given a deterministic anytime algorithm $\mathcal{A}$, let $q_{\mathcal{A}}(x,t)$ be the quality of results produced by $\mathcal{A}$ with input x and computation time t; let $q_{\mathcal{A}}(t)$ be the expected quality of results with computation time t; and let $p_{\mathcal{A},t}(q)$ be the probability (density function in the continuous case) that $\mathcal{A}$ with computation time t produces results of quality q.

The most informative type of performance profile used in this work is the performance distribution profile defined below:

Definition 2.1. The *performance distribution profile* (*PDP*) of an algorithm $\mathcal{A}$ is a function $D_{\mathcal{A}} : \mathbb{R}^+ \rightarrow Pr(\mathbb{R})$ that maps computation time to a probability distribution over the quality of the results.

An obvious simplification of the PDP is the expected performance profile (EPP), as used by Boddy and Dean [2] and by Horvitz [14]:

Definition 2.2. The *expected performance profile* (*EPP*) of an algorithm $\mathcal{A}$ is a function $E_{\mathcal{A}} : \mathbb{R}^+ \rightarrow \mathbb{R}$ that maps computation time to the expected quality of the results.

Note that $E_{\mathcal{A}}(t)$ can be calculated directly when the expected quality of the algorithm can be determined for each time allocation or it can be estimated by averaging the actual quality achieved over many problem instances (as shown in Eq. (1).

$$E_{\mathcal{A}}(t) = \sum_{q} p_{\mathcal{A},t}(q)q = \sum_{x} Pr(x)q_{\mathcal{A}}(x,t). \tag{1}$$

For any summary description of component algorithms, it is important to understand how the summary description of a composite system can be derived from the descriptions of its components. Suppose, for example, that for any particular input to a two-component system, the output quality is some function f of the qualities $q_1(x,t)$ and $q_2(x,t)$ achieved by the components. Unfortunately, it is generally the case that

$$f(E_x(q_1(x,t)), E_x(q_2(x,t))) \neq E_x(f(q_1(x,t), q_2(x,t))).$$

Hence the EPP of the composed system cannot be recovered easily from the EPPs of the components. EPPs are therefore most useful when the variance of the original PDPs is small, so that the error associated with composition of EPPs is also small. In the special case where the variance of the distribution is zero (or infinitesimal), the anytime algorithm is said to have a *fixed performance*. For such algorithms, an expected performance profile offers a complete, accurate description of performance.

Definition 2.3. The *performance interval profile* (*PIP*) of an algorithm $\mathcal{A}$ is a function $I_{\mathcal{A}} : \mathbb{R}^+ \rightarrow \mathbb{R} \times \mathbb{R}$ that maps computation time to the upper and lower bounds of the quality of the results.

Note that if $I_{\mathcal{A}}(t) = [L, U]$ then:

$$\forall x: \quad L \leqslant q_{\mathcal{A}}(x,t) \leqslant U. \tag{2}$$

Performance interval profiles offer a representation that is both compact and easy to manipulate. From the lower bounds on the qualities of the results of two algorithms, one can normally find a lower bound on the quality of their combined result. Hence, when a compact representation is preferred and the variance of the distribution is wide, performance interval profiles are useful.

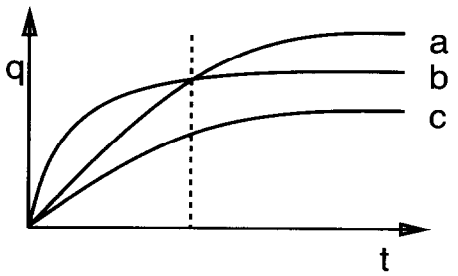

Fig. 2. Dominance relations among profiles: a and b dominate c, but neither of a and b dominates the other.

In order to define optimal compilation, we will also need a notion of *dominance* among profiles.

Definition 2.4. Let $\mathcal{A}$ and $\mathcal{B}$ be two anytime algorithms that solve the same problem, then $\mathcal{B}$ is said to *dominate* $\mathcal{A}$ ($\mathcal{B} \succ \mathcal{A}$) if for every input x and every time allocation t:

$$\forall x \; \forall t: \quad q_{\mathcal{B}}(x,t) \geqslant q_{\mathcal{A}}(x,t).$$

The relationship of dominance between anytime algorithms is a partial order. Given two anytime algorithms that solve a certain problem, it is possible that neither of them dominates the other (see Fig. 2).

2.2.2. Conditional performance profiles

It may happen that the performance of the algorithm depends significantly on the nature of the inputs, in which case the PDP will be too coarse for general use. If the input dependence can be attributed to a small set of features, one can use a *conditional performance profile* by partitioning the input domain into classes and storing a separate profile for each input class. The partitioning can be done using any attribute of the input that may influence performance, such as size or a complexity measure. Input classes of similar performance can also be derived automatically using Bayesian statistics by programs such as Autoclass [3].

In this paper, we consider conditioning on the input quality. A conditional performance profile therefore consists of a mapping from input quality and run-time to probability distribution of output quality:

Definition 2.5. The *conditional performance profile* (*CPP*) of an algorithm $\mathcal{A}$ is a function $C_{\mathcal{A}} : \mathbb{R} \times \mathbb{R}^{+} \rightarrow Pr(\mathbb{R})$ that maps input quality and computation time to a probability distribution over the quality of the results.

A CPP can also be seen as a family of PDPs, each for a different input quality and denoted by $C_{\mathcal{A},q}$. This leads to the graphical representation shown in Fig. 3. Each curve in the figure represents an expected performance profile for a particular input quality.

The information in the CPP is essential to the compilation process since the allocation of time to a certain module affects not only the quality of the result of that module but

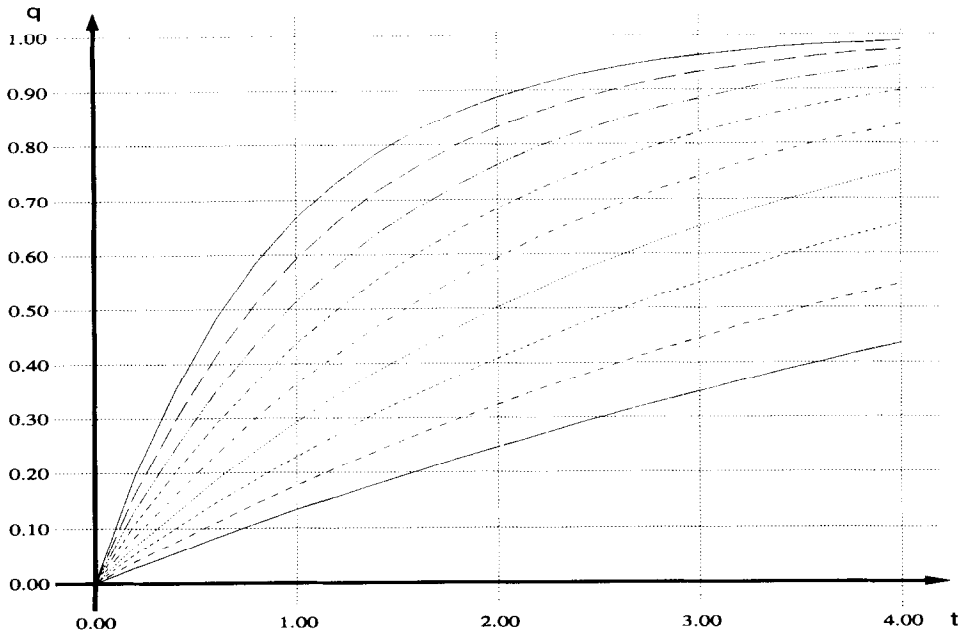

Fig. 3. Graphical representation of a conditional performance profile.

also the quality of the results of any module that uses that result. The CPP thus provides enough information to characterize the performance of any module in such a way that it can be combined optimally with other modules without "looking inside".

Definition 2.6 (*Input monotonicity*). A CPP for algorithm $\mathcal{A}$ exhibits input monotonicity if and only if

$$\forall p, q: \quad p > q \Rightarrow C_{\mathcal{A},p} \succ C_{\mathcal{A},q}.$$

That is, as the input quality improves, so should the performance profile. This is a very natural property, and also very useful as we show below.

2.2.3. Acquiring and representing performance profiles

Performance profiles can sometimes be calculated by algorithm analysis. For example, in many iterative algorithms, such as Newton's method, the error in the result is bounded by a function that depends on the number of iterations. In such cases, the performance profile can be calculated once the run-time of a single iteration is determined. In general, however, such structural analysis of the code is hard because the improvement in quality in each iteration and its run-time may be unpredictable. To overcome this difficulty, a general *simulation* method can be used. It is based on gathering statistics on the performance of the algorithm on randomly generated problem instances. Ideally, the statistics are gathered for the same population of instances as will appear when

the algorithm is deployed. This can be ensured by learning the profiles during actual operation.

Performance profiles can be represented either by a closed form or as a table of discrete entries. Since performance profiles are normally monotone functions of time, they can be approximated using a simple family of functions. Once the simulation data is gathered, the performance information can be derived by various curve fitting techniques. For example, Boddy and Dean [2] used the function: $Q(t) = 1 - e^{-\lambda t}$ to model the expected performance of an anytime planner. Performance distribution profiles can be approximated by applying a similar method to a family of distributions. For example, if the normal distribution is used, one can apply curve fitting techniques to approximate the mean and variance of the distribution as a function of time.

The advantage of using a closed-form representation of performance profiles is that optimization of time allocations can be performed for a general parameterized family of profiles, using straightforward calculus techniques. The results of such compilation can be used each time members of that family are compiled. Closed-form representation has two major disadvantages: (1) fitting a closed-form approximation to real data may involve a large error; and (2) it is hard to maintain closure under the compilation operation. The closure property requires that the result of compilation of two (or more) performance profiles that belong to a certain family be a member of the same family, or at least that it be approximable by a function in that family. The disadvantages of the closed-form representation led us to use a more flexible, discrete representation.

The discrete representation of performance profiles is based on a table that specifies the discrete probability distribution over quality for a range of time allocations. For this purpose, the complete range of qualities has to be divided into discrete qualities $q_1, \ldots, q_n$. The entry i, j in the table represents the discrete probability that with time allocation t_i the actual output quality q is in the range $[q_j - \delta, q_j + \delta]$. The size of the table is a system parameter that controls the accuracy of performance information. Linear interpolation is used to find the quality when the run-time does not match exactly one of the table entries.

2.3. *Interruptible versus contract algorithms*

We make an important distinction between two types of anytime algorithms called *interruptible* algorithms and *contract* algorithms. Interruptible algorithms produce results of the "advertised quality" even when interrupted unexpectedly. Contract algorithms, although capable of producing results whose quality varies with time allocation, must be given a particular time allocation in advance. If a contract algorithm is interrupted at any time shorter than its contract time, it may yield no useful results. Both interruptible and contract algorithms have been used in the past. Dean and Boddy's [7] definition of anytime algorithms refers to the interruptible case. Techniques such as depth-limited search and alpha-beta search, on the other hand, are more suited for contract algorithms. Although they can produce a suitable result for any given effort limit, they may return meaningless results if interrupted before completion.

In general, every interruptible algorithm is trivially a contract algorithm, but the converse is not true. Intuitively, one tends to think about anytime algorithms as interruptible,

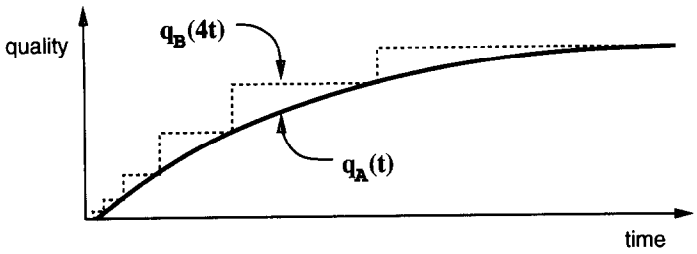

Fig. 4. Performance profiles of interruptible and contract algorithms.

yet the greater freedom of design makes it easier to construct contract algorithms than interruptible ones. In the case of functional composition, as illustrated by the real-time medical system mentioned above, it is possible to allocate a fixed contract time optimally between the two components. This results, however, in a contract algorithm since interrupting the system during diagnosis leaves one with no treatment recommendation at all. This is the case even if the individual components are themselves interruptible. Thus naïve composition destroys interruptibility. This problem is solved by the following reduction theorem [30]:

Theorem 2.7 (Reduction). *For any contract algorithm $\mathcal{A}$, an interruptible algorithm $\mathcal{B}$ can be constructed such that for any particular input $q_{\mathcal{B}}(4t) \geqslant q_{\mathcal{A}}(t)$.*

Proof. Construct $\mathcal{B}$ by running $\mathcal{A}$ repeatedly with exponentially increasing time limits. If interrupted, return the best result generated so far. Let the sequence of run-time segments be $\epsilon, 2\epsilon, \ldots, 2^i\epsilon, \ldots$, and assume that the time overhead of the code required to control this loop can be ignored. Note also that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$. The worst case situation occurs when $\mathcal{B}$ is interrupted after almost $(2^n - 1)\epsilon$ time units, just before the last iteration terminates and the returned result is based on the previous iteration with a run-time of $2^{n-2}\epsilon$ time units. Since $(2^n - 1)/2^{n-2} < 4$, the factor of 4 results. If one replaces the multiplier of time intervals by α, one gets a time ratio of $(\alpha^n - 1)/(\alpha^{n-1} - \alpha^{n-2})$. The lower bound of this expression is 4, for $\alpha = 2$, hence 2 is the optimal multiplier under this strategy. □

Fig. 4 shows a typical performance profile for the contract algorithm $\mathcal{A}$, and the corresponding performance profile for the constructed interruptible algorithm $\mathcal{B}$, reduced along the time axis by a factor of 4. As an example, consider the application of this construction method to Korf's RTA*, a contract algorithm. As the time allocation is increased exponentially, the algorithm will increase its depth bound by a constant; the construction therefore generates an iterative deepening search automatically.

2.4. *Programming techniques for anytime algorithms*

The development of elementary anytime algorithms does not require a radical change in programming methodologies. Many existing programming and automated reasoning techniques produce useful anytime algorithms: search techniques such as iterative deep-

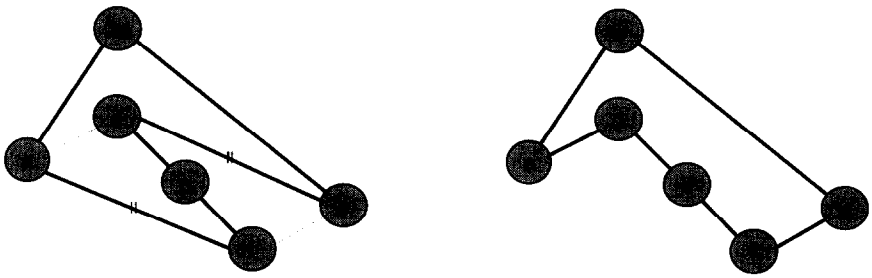

Fig. 5. The operation of randomized tour improvement.

ening; asymptotically correct inference algorithms such as approximate query answering [9,32], bounded cutset conditioning (see [14]), and variable precision logic [24]; various greedy algorithms (see [2]); iterative methods such as Newton's method; adaptive algorithms such as PAC learning algorithms or neural networks; randomized methods such as Monte Carlo algorithms or fingerprinting techniques [17]; and the use of optimal meta-level control of computation [28]. We conclude this section with an example of a particular anytime algorithm and its performance profile.

Example [The Traveling Salesman Problem] The *Traveling Salesman Problem* (TSP) involves a salesman that must visit n cities. If the problem is modeled as a complete graph with n vertices, the solution becomes a *tour*, or Hamiltonian cycle, visiting each city exactly once, starting and finishing at the same city. The cost function, $Cost(i, j)$, defines the cost of traveling directly from city i to city j. (The cost is not necessarily the Euclidean distance.) The problem is to find an optimal tour, that is, a tour with minimal total cost. The TSP is known to be NP-complete [10], hence it is hard to find an optimal tour when the problem includes a large number of cities. *Iterative improvement* algorithms can find a good approximation to an optimal solution, and naturally yield an interruptible anytime algorithm.

The anytime traveling salesman algorithm is a randomized algorithm that repeatedly tries to perform a tour improvement step [20,22]. In the general case of tour improvement procedures, r edges in a feasible tour are exchanged for r edges not in that solution as long as the result remains a tour and the cost of that tour is less than the cost of the previous tour. The simplest case is when $r = 2$. Fig. 5 demonstrates one step of tour improvement. An existing tour, shown in part (a), visits the vertices in the following order: *a, b, c, d, e, f*. The algorithm selects two random edges of the graph, (c, d) and (f, a) in this example, and checks whether the following condition holds:

$$Cost(c, f) + Cost(d, a) < Cost(c, d) + Cost(f, a). \tag{3}$$

If this condition holds, the existing tour is replaced by the new tour, shown in part (b), *a, b, c, f, e, d*. The improvement condition guarantees that the new path has a lower cost. The algorithm starts with a random tour that is generated by simply taking a random ordering of the cities. Then the algorithm tries to reduce the cost by a sequence

```
ANYTIME-TSP(V, iter)
1   Tour ← INITIAL-TOUR(V)
2   cost ← COST(Tour)
3   REGISTER-RESULT(Tour)
4   for i ← 1 to iter
5     e1 ← RANDOM-EDGE(Tour)
6     e2 ← RANDOM-EDGE(Tour)
7     δ ← COST(Tour) − COST(SWITCH(Tour, e1, e2))
8     if δ > 0 then
9       Tour ← SWITCH(Tour, e1, e2)
10      cost ← cost − δ
11      REGISTER-RESULT(Tour)
12  SIGNAL(TERMINATION)
13  HALT
```

Fig. 6. The anytime traveling salesman algorithm.

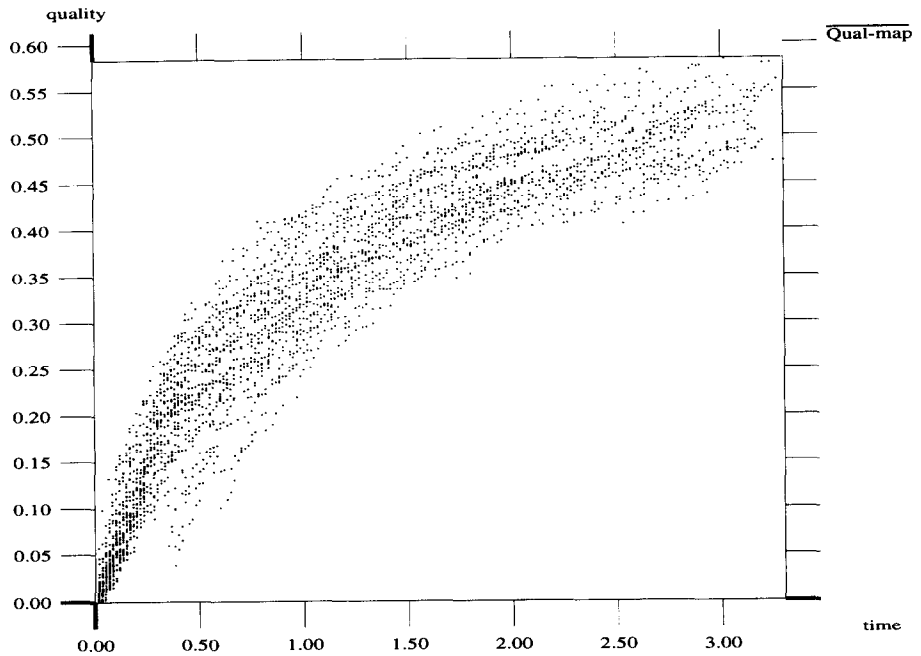

Fig. 7. The quality map of the TSP algorithm.

of random improvements. The result is an interruptible anytime algorithm, as shown in Fig. 6. Note that the algorithm has a generic design that includes an initial step to generate and register the first result followed by a loop containing an improvement step. The compiled code handles an interrupt by returning the most recently registered result.

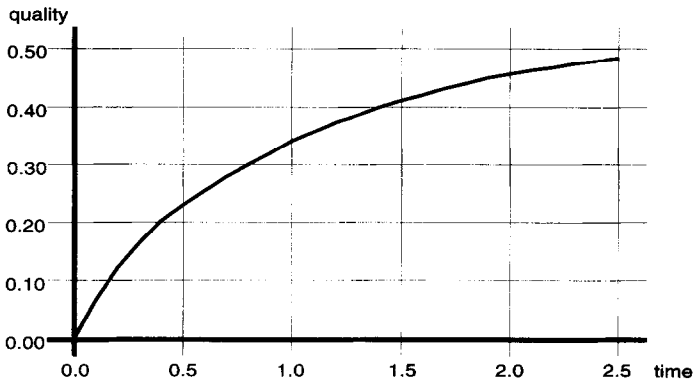

Fig. 8. The expected performance profile of the TSP algorithm.

The *iter* argument indicates the maximum number of iterations but execution can be interrupted by the monitor at an earlier point.

Fig. 7 shows the *quality map* of the algorithm, which summarizes the results of many activations with randomly generated input instances (including 50 cities). Each point (t, q) represents an instance for which quality q was achieved with run-time t. The quality of results in this experiment measures the percentage of tour length reduction with respect to the initial tour. These statistics form the basis for the construction of the performance profile of the algorithm. The resulting expected performance profile is shown in Fig. 8.

3. Compilation of anytime algorithms

We now turn from the examination of individual anytime algorithms to the problem of building large systems using anytime algorithms as components. The compilation process, illustrated in Fig. 9, plays a central role in the solution to this problem.

The input to the compiler is a compound anytime module, that is, a module composed of several elementary anytime algorithms. The primitive programming language constructs that are used to define compound modules can vary from a small set of simple constructs to a rich programming language [34]. The choice of language primitives determine the feasibility and complexity of the compilation problem. Compound modules do not include time allocation code and hence they are not readily executable. In addition to the compound module, the compiler's input includes the performance profiles of the elementary anytime algorithms. The result of the compilation process is an executable anytime module that consists of a compiled version of the original module, a pre-defined run-time monitor, and the performance profile of the system that may include some auxiliary time allocation information. The compiled module includes code to control the activation of the elementary components with an appropriate time allocation.

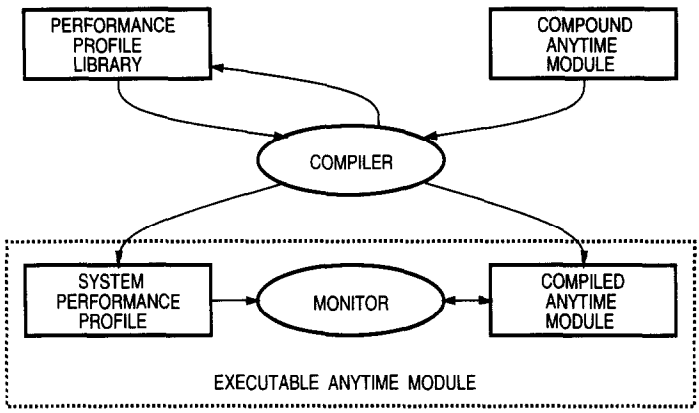

Fig. 9. Compilation and monitoring.

Optimal scheduling of the elementary components may also require run-time monitoring. The problem addressed by the monitor is similar to the *deliberation scheduling* problem introduced by Dean and Boddy in [7]. Previous solutions to the problem included only a small set of algorithms characterized by non-conditional performance profiles. In this work we have studied the composition and monitoring of an arbitrary number of different algorithms characterized by conditional performance profiles. We found that the complexity of the compilation process is largely determined by the choice of a run-time monitoring scheme. Active monitoring, that revises the allocation the the components while the system is active, is discussed in [34]. To simplify the discussion here, we assume that time allocation to the components is determined prior to the activation of the system.

3.1. Aspects of the compilation problem

The solution to the compilation problem depends on a number of factors that characterize the inputs and the outputs of the process. The main aspects of the problem are described below:

(1) *Program structure.* The structure of a compound anytime module is a primary factor that determines the complexity of its compilation. Some programming structures, such as sequencing, are easier to handle, while others, such as recursive function calls, are quite difficult to compile.

(2) *Type of performance profiles.* The type of performance profiles and their representation also influence the compilation process. Highly informative performance profiles, such as the performance distribution profile, are more difficult to compile and manipulate. The complexity of the compilation is increased due to the complexity of the representation and the requirement that the resulting performance profile provides the same level of information.

(3) *Type of anytime algorithms.* The type of algorithm used as input to the compiler and the desired type of the resulting algorithm have a direct effect on the

compilation process. Contract algorithms are normally easier to construct both as elementary and as compound algorithms. Interruptible algorithms are more complicated. One can, of course, construct first a contract algorithm and then use the result of Theorem 2.7 to make it interruptible. However, with some programming structures it is advantageous to generate an interruptible algorithm directly and avoid the constant slowdown of the reduction theorem.

(4) *Type of monitoring.* Anytime computation can be controlled using either passive or active monitoring. Passive monitoring means that meta-level scheduling decisions are made *before* the activation of the anytime algorithms. Elementary algorithms are activated as contract algorithms only and their run-time cannot be modified before the termination of the contract. Obviously, the assumption of passive monitoring limits the capability to optimize the performance profile of a system, but it also simplifies the compilation problem. With active monitoring, time allocation decisions may be made after the activation of the system, in response to the *actual* rather than *expected* performance of the components.

(5) *Quality of intermediate results.* With both interruptible and contract anytime algorithms, an active monitor can examine the quality of intermediate results in order to modify the allocation of the remaining time. However, this requires a capability to determine the *actual* quality of intermediate results. The quality of intermediate results may be a simple aspect that can be quickly calculated. For example, in the case of a bin packing program whose quality function is the proportion of the container space filled with packages, the quality of an intermediate result can be easily calculated. In other cases, such as a chess playing program, the quality of a recommended move is not apparent from the move itself. Hence, the capability to determine the quality of intermediate results is an important factor in compilation and monitoring.

3.2. Compilation examples

As a simple example of compilation, consider the composition of two anytime algorithms. Suppose that one algorithm takes the input and produces an intermediate result. This result is then used as input to another anytime algorithm which, in turn, produces the final result. Many systems can be implemented by a composition of a sequence of two or more algorithms. We will examine two particular systems. The first is a repair system whose elementary performance profiles are represented using a closed form. The second is a path planning system whose performance profiles are represented using the discrete tabular approach.

3.2.1. Composition of diagnosis and treatment planning

Consider an automated repair system that is composed of two anytime algorithms: diagnosis and treatment planning. The system can be represented by the following expression:

$$Output \leftarrow \text{TREATMENT}(\text{DIAGNOSIS}(Input)).$$

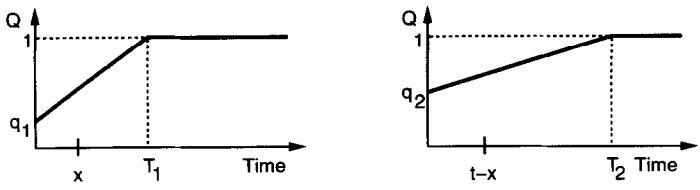

Fig. 10. Performance profiles of DIAGNOSIS and TREATMENT.

The input to DIAGNOSIS is a set of symptoms for which a diagnosis is computed. This diagnosis is used as input to TREATMENT that produces the final output—a treatment plan. Fig. 10 shows the linear performance profiles of the elementary anytime algorithms. They start with an arbitrary initial quality q_i (that may be zero) and reach the maximal quality of 1 at time T_i. Hence they can be represented by:

$$Q_1(t) = q_1 + \alpha_1 t \quad (0 \leqslant t \leqslant T_1),$$
$$Q_2(t) = q_2 + \alpha_2 t \quad (0 \leqslant t \leqslant T_2).$$

The quality of DIAGNOSIS, Q_1, reflects the probability that the diagnosis is correct. Similarly, the quality of TREATMENT, Q_2, reflects the probability that the treatment plan repairs the problem *given* that the diagnosis is correct. Assuming that the qualities of the two modules are independent, we can express the overall quality by the product of the qualities of the two modules. Our goal is to compile the best contract algorithm for the complete system. In other words, the compilation process has to create the following mappings:

$$\mathcal{T} : \mathbb{R}^+ \to \mathbb{R}^+ \times \mathbb{R}^+, \tag{4}$$

$$\mathcal{PP} : \mathbb{R}^+ \to [0, 1]. \tag{5}$$

The first mapping specifies for each total allocation the amount of time that should be allocated to each algorithm so as to maximize the output quality.[1] The second mapping is the performance profile of the composed algorithm based on optimal time allocation. For each total allocation, t, the compiler has to find the optimal allocation, x, to the first algorithm (which implies allocation $t - x$ to the second algorithm) such that the overall quality $Q(x)$ is maximal.

Theorem 3.1. *Given the performance profiles of the input modules, the optimal time allocation mapping is:*

$$\mathcal{T} : t \to \left(\frac{1}{2}\left(t - \frac{q_1}{\alpha_1} + \frac{q_2}{\alpha_2}\right), \frac{1}{2}\left(t + \frac{q_1}{\alpha_1} - \frac{q_2}{\alpha_2}\right)\right). \tag{6}$$

Proof. Since the overall output quality is:

$$Q(x) = -\alpha_1\alpha_2 x^2 + (\alpha_1\alpha_2 t - q_1\alpha_2 + q_2\alpha_1)x + q_1 q_2 + q_1\alpha_2 t. \tag{7}$$

[1] Only the appropriate allocation to the first component is really necessary because the allocation to the second is simply the remaining time.

the maximal quality is achieved when $\partial Q/\partial x = 0$, or when:

$$-2\alpha_1\alpha_2 x + \alpha_1\alpha_2 t - q_1\alpha_2 + q_2\alpha_1 = 0. \tag{8}$$

The solution of this equation yields the above allocation. □

A trivial correction is needed to cover boundary conditions (since allocation to DIAGNOSIS should be in $[0, T_1]$ and to TREATMENT in $[0, T_2]$): (1) if an algorithm gets more run-time than is necessary for its completion, then the extra time should be allocated to the other algorithm (or ignored when both algorithms terminate); and (2) if the time allocation to one algorithm is negative, then all the available time should go to the other algorithm.

3.2.2. Composition of sensing and path planning

Consider a robot navigation system that is composed of two anytime algorithms: visual sensing and path planning. The system can be represented by the following expression:

Output ← PATH-PLAN(*Start*, *Goal*, GET-DOMAIN-DESCRIPTION(*Input*)).

The input to GET-DOMAIN-DESCRIPTION is raw data from a visual sensor from which the module constructs an approximate map of the robot's local environment. This map is used as input to PATH-PLAN that produces the final output—a path from *Start* to *Goal*. The actual implementation of these anytime modules is described in [37]. Fig. 11 shows the performance profiles.

The domain is represented as a matrix of elementary positions each of which can be either free or occupied by an obstacle. The quality of GET-DOMAIN-DESCRIPTION reflects the probability that an elementary domain position would be wrongly identified, that is, identified as free space while actually blocked by an obstacle or vice versa. In Fig. 11, T_a is the minimal amount of time needed for the module to produce an initial domain description with quality Q_a. For a run-time t, $T_a \leqslant t \leqslant T_b$, the quality of GET-DOMAIN-DESCRIPTION improves from Q_a to the maximal quality Q_b.

Path planning is performed using a coarse-to-fine search algorithm (similar to that of Lozano-Pérez and Brooks [23]) that allows for unresolved path segments. In order to make it an anytime algorithm, we vary the abstraction level of the domain description. This allows the algorithm to find a feasible plan quickly, and then repeatedly refine it by replanning a segment of the plan in more detail. The quality of a plan is the ratio between the length of the shortest path and the path that the robot follows when it uses the abstract plan. To capture the dependency of the quality of planning on the quality of sensing, we used a conditional performance profile.

Performance profiles in this application were represented using the discrete tabular approach. Using this representation, the compilation of the two modules becomes a discrete optimization problem that we solved using a simple search algorithm. Fig. 12 shows the resulting performance profile. Also shown in that figure are the performance profiles of two other modules: MIN, that allocates to GET-DOMAIN-DESCRIPTION a minimal amount of time, T_a, and MAX, that allocates a maximal amount of time, T_b. The compiled performance profile is superior to both. It is closer to MIN with small allocations of time and is closer to MAX in the limit.

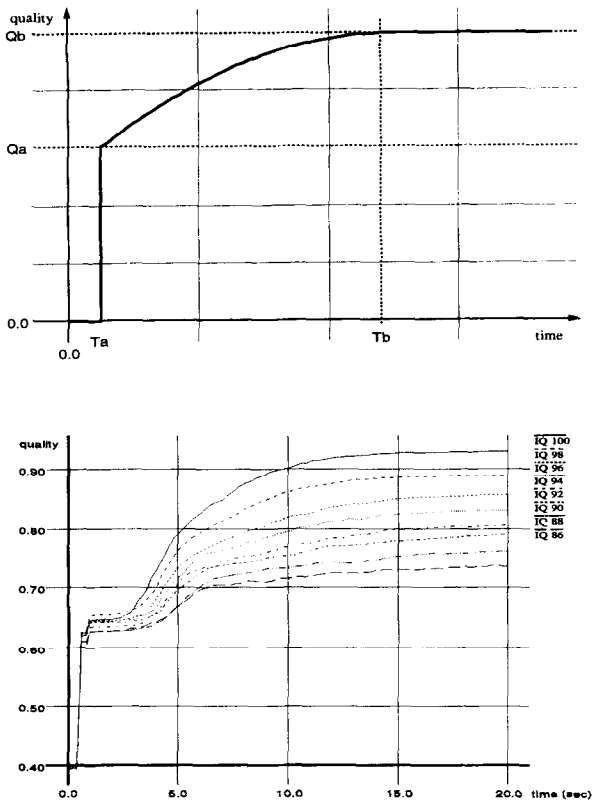

Fig. 11. (a) Expected performance profile of GET-DOMAIN-DESCRIPTION. (b) Conditional performance profile of PATH-PLAN, given input quality between 0.86 and 1.0.

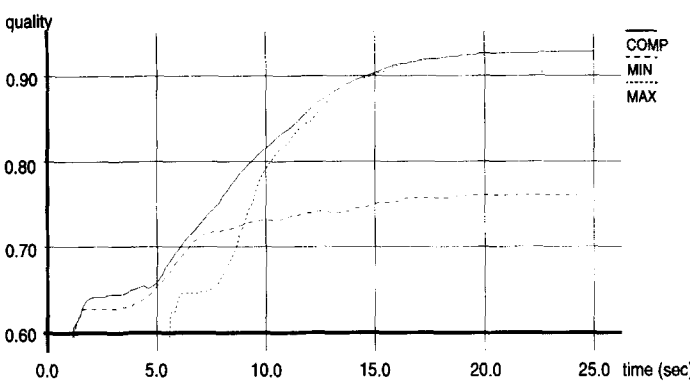

Fig. 12. The compiled performance profile for the composed system (COMP). The profiles labeled MIN and MAX show the result of minimal and maximal allocations to the vision component.

The above two examples demonstrate several general issues in compilation. When performance profiles are represented using a certain formula, as in the first example, the compilation problem involves solving a differential equation. The complexity of the equation, in terms of both size and number of variables, grows as a function of the number of elementary algorithms that are compiled. If a discrete tabular representation is used, then the compilation problem becomes a search problem in a discrete domain whose size grows exponentially with the number of modules. The problem of exponential growth in the complexity of compilation is addressed in the next section.

4. Compilation of functional expressions

We now turn to a more formal analysis of a general class of compilation problems, namely the family of programs created by functional composition of anytime algorithms. In functional composition each expression to be compiled is composed of an anytime function whose arguments may be either input variables or another expression created by functional composition. In the case of contract algorithms, the compilation task involves finding for each total allocation t, the best way to schedule the components so as to optimize the expected quality of the result of the complete expression.

Let $\mathcal{F}$ be a set of anytime functions. To simplify the discussion, assume that all function parameters are passed by value and that functions have no side-effects (as in pure functional programming). Let $\mathcal{I}$ be a set of input variables. The notion of a functional expression is defined as follows:

Definition 4.1. A *functional expression* over $\mathcal{F}$ with input $\mathcal{I}$ is:

(1) an input variable $i_j \in \mathcal{I}$, or

(2) an expression $f(g_1, \ldots, g_n)$ where $f \in \mathcal{F}$ and each g_j is a functional expression.

Each function $f \in \mathcal{F}$ has a fixed conditional performance profile associated with it that specifies the quality of its output as a function of time allocation and input quality.

Fig. 13 shows two possible graphical representations of the functional expression:

$$F(x) = E(D(B(A(x)), C(A(x)))).$$

The first representation is a tree constructed in the following way:

(1) If e is an input variable i_j, then it is represented by a leaf node i_j.

(2) If $e = f(g_1, \ldots, g_n)$, then it is represented by a tree whose root node is f and whose main subtrees are the trees representing $g_1, \ldots, g_n$.

The second representation is a directed acyclic graph (DAG) constructed in the following way:

(1) If e is an input variable i_j, then it is represented by a leaf node i_j.

(2) If $e = f(g_1, \ldots, g_n)$, then it is represented by a DAG that includes a node f and directed arcs from the (roots of the) DAGs representing $g_1, \ldots, g_n$ to f.

Notice that the DAG representation requires only one DAG to represent all the copies of a repeated subexpression, while the tree representation requires multiple copies of subtrees for repeated subexpressions. When a functional expression has no repeated

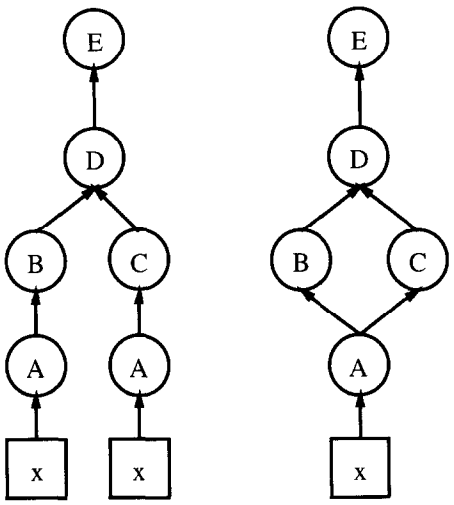

Fig. 13. Graph representation of functional expressions.

subexpressions, its tree and DAG representations are identical and its compilation is simplified.

4.1. The complexity of compilation

In this section we will analyze the complexity of compilation of functional expressions and show that the general problem is NP-complete in the strong sense. A relaxed version of the problem, that excludes repeated subexpressions, will be shown to be pseudo-polynomial.

The compilation problem is normally defined as an optimization problem, that is, a problem of finding a schedule for a set of components that yields maximal output quality. But in order to prove NP-completeness results, it is more convenient to refer to the decision problem variant of the compilation problem. This decision problem is stated as follows: given a functional expression e, the conditional performance profiles of its components, and a total allocation B, does there exist a schedule of the components that yields output quality greater than or equal to K? We refer to this decision problem as the problem of global compilation of functional expressions, or GCFE. The first complexity result asserts the following:

Theorem 4.2. *The GCFE problem is* NP*-complete in the strong sense.*

Proof. The GCFE problem is clearly NP since, given a particular allocation to the components, it is easy to determine in linear time the output quality of the expression. Hence, the verification problem is polynomial and the decision problem is NP. The rest of the proof is by transformation from the PARTIALLY ORDERED KNAPSACK

problem, an NP-complete problem in the strong sense [10] defined as follows:

- *Instance.* Finite set U, partial order $\prec$ on U, for each $u \in U$ a size $s(u) \in \mathbb{Z}^+$ and a value $v(u) \in \mathbb{Z}^+$, and positive integers B and K.
- *Question.* Is there a subset $U' \subseteq U$ such that if $u \in U'$ and $u' \prec u$, then $u' \in U'$, and such that $\sum_{u \in U'} s(u) \leqslant B$ and $\sum_{u \in U'} v(u) \geqslant K$?

An instance of the PARTIALLY ORDERED KNAPSACK problem can be directly transformed into a DAG representing a corresponding functional expression. To describe the construction of the DAG, we must first define the notion of a maximal element in a partially ordered set.

Definition 4.3. An element $u \in U$ is a *maximal element* of U if there is no other element $u' \in U$ such that $u \prec u'$.

The notion of a minimal element is defined in an analogous way. Every partially ordered set has at least one maximal element and at least one minimal element. Now, the construction of the DAG is defined as follows. For each $u \in U$ the DAG will contain a corresponding computational node. A direct arc goes from u_1 to u_2 if and only if u_1 is a maximal element of the set $\{u \mid u \prec u_2\}$ of all elements smaller than u_2. In addition, the DAG has a "root" node r with a directed arc from every other node $u \in U$ to r. The conditional performance profile of a node $u \in U$ is:

$$Q_u(q_1, \ldots, q_n, t) = \begin{cases} v(u), & \text{if } t \geqslant s(u) \text{ and } \forall i : q_i > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

where $q_1, \ldots, q_n$ are the qualities of the nodes that have a directed arc to u. If there is no such node, that is, if u is a minimal element of U, then its performance profile is:

$$Q_u(t) = \begin{cases} v(u), & \text{if } t \geqslant s(u), \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

The conditional performance profile of r is defined as follows:

$$Q_r(q_1, \ldots, q_k, t) = \sum_{i=1}^{k} q_i. \tag{11}$$

Finally, the overall output quality Q_{out} is defined as the quality of the root node, r.

It is easy to see that the construction of the DAG can be accomplished in polynomial time. All that is left to show is that the answer to the PARTIALLY ORDERED KNAPSACK problem is "yes" if and only if the answer to the corresponding GCFE problem is "yes".

If the answer to the GCFE problem is positive (with contract time B and minimal output quality K), then define U' as the set of nodes $u' \in U$ whose "output quality" in the DAG is positive. The sum of the output qualities of all the modules, except the root, must be at least K. Each module can only contribute its value to the output quality when its allocation is at least its size. In addition, the output quality of an internal node of the DAG is "enabled" only when all its inputs have positive quality, that is, all the

elements smaller than it are included. Therefore the condition that $u' \in U'$ when $u \in U'$ and $u' \prec u$ is satisfied. Finally, since the total allocation is B, $\sum_{u \in U'} s(u) \leqslant B$, and since the output quality is at least K, $\sum_{u \in U'} v(u) \geqslant K$, the answer to the PARTIALLY ORDERED KNAPSACK problem is also positive.

If the answer to the PARTIALLY ORDERED KNAPSACK problem is positive (with knapsack size B and minimal value K), then simply allocate to each computational node $u' \in U'$ an amount of time equal to its size. The definition of the PARTIALLY ORDERED KNAPSACK problem and the transformation to the DAG guarantee that the output quality of each u' would be equal to its value $s(u')$. Hence a minimal output quality of K is guaranteed and the answer to the GCFE problem is also positive.

Since the PARTIALLY ORDERED KNAPSACK problem is NP-complete in the strong sense, and since the above transformation is polynomial, the GCFE problem is NP-complete in the strong sense. □

We now turn to the analysis of a relaxed case of the compilation problem, referred to as tree-structured GCFE. In this case, no repeated subexpressions are allowed and as a result the DAG representation becomes a directed tree. We show that the tree-structured GCFE is NP-complete.

Theorem 4.4. *The tree-structured GCFE problem is* NP*-complete.*

Proof. As in the case of the GCFE problem, the verification problem is polynomial and the problem is therefore NP. The rest of the NP-completeness proof is by transformation from the KNAPSACK problem [10], defined as follows:

- *Instance.* Finite set U, for each $u \in U$ a size $s(u) \in \mathbb{Z}^+$ and a value $v(u) \in \mathbb{Z}^+$, and positive integers B and K.
- *Question.* Is there a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leqslant B$ and $\sum_{u \in U'} v(u) \geqslant K$?

An instance of the KNAPSACK problem can be transformed into a tree-structured GCFE problem by constructing a binary tree whose leaves are the elements of U. Each element $u \in U$ corresponds to one leaf of the tree (one can add leaf nodes of zero size and value to make the number of leaves an exact power of 2). The performance profile of each leaf node is:

$$Q_u(t) = \begin{cases} v(u), & \text{if } t \geqslant s(u), \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

Now, $O(|U|)$ internal nodes are added to construct a complete binary tree. The conditional performance profile of each internal node, w, is the sum of the qualities of its left and right branches:

$$Q_w(q_1, q_2, t) = q_1 + q_2. \tag{13}$$

Note that internal nodes of the tree do not consume any computation time. The output quality, Q_{out}, is the quality of the root node which is the sum of the values of all the the elements of U whose allocation exceeds their size.

It is easy to see that the construction of the tree can be accomplished in polynomial time. To complete the proof, we need to show that the answer to the KNAPSACK problem is "yes" if and only if the answer to the corresponding tree-structured GCFE problem is "yes". This is trivially true when one sets the contract time to B and the minimal output quality to K. The argument is similar to the previous proof. We conclude that the tree-structured GCFE problem is NP-complete. □

The KNAPSACK problem itself is pseudo-polynomial. In fact, the problem can be solved by a simple dynamic programming algorithm. This raises the question of whether the compilation problem of tree-structured expressions is also pseudo-polynomial. The next section identifies the conditions under which the answer to this question is positive.

4.2. *Local compilation*

Local compilation is the key mechanism in our model to cope with the exponential complexity of global compilation. The idea is to replace a single, complex optimization problem with a set of simple optimization problems whose number grows *linearly* with the size of the program being compiled. If these local optimization problems can be solved in polynomial time, then the total amount of work becomes polynomial.

Definition 4.5. *Local compilation* is the process of optimizing the quality of the output of each programming construct by considering only the performance profiles of its immediate subcomponents.

Local compilation solves the same type of problem as global compilation except for the fact that its scope is limited to one programming structure at a time. While global compilation derives directly the best time allocation to *all* the elementary components, local compilation computes the best time allocation to the immediate subcomponents, treating them as if they were elementary anytime algorithms. If a subcomponent is not elementary, then its performance profile is derived using local compilation as well.

A fundamental question regarding local compilation is the relationship between its result and the result of global compilation. Local compilation is said to be optimal with respect to a particular program structure if it always achieves a globally optimal expected performance. Our first goal in this section is to prove the optimality of local compilation of tree-structured functional expressions under the input monotonicity assumption. Without loss of generality, we will consider binary functions only and assume that the functional expression is a complete binary tree. The leaves of the tree are functions that take input variables as inputs and the internal nodes are functions that take functional expressions as inputs.

Let $f_{i,j}$ denote the jth function on the ith level of the tree. The root node is denoted accordingly by $f_{0,0}$. If the tree is of depth n, then the nodes corresponding to $f_{n,0}, \ldots, f_{n,2^n-1}$ are leaf nodes whose inputs are input variables. For any other node $f_{i,j}, 0 \leqslant i \leqslant n-1, 0 \leqslant j \leqslant 2^i - 1$, the inputs are: $f_{i+1,2j}$ and $f_{i+1,2j+1}$ as shown in Fig. 14.

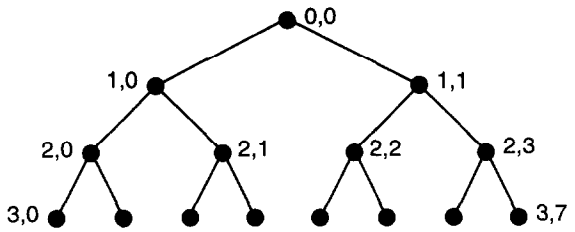

Fig. 14. Tree representation of a functional expression.

Corresponding to each node of the binary tree is a conditional performance profile $Q_{i,j}(q_1, q_2, t)$ which characterizes the output quality for that node as a function of its input qualities, q_1 and q_2, and time allocation t.

Given a functional expression e of depth n, and a particular input quality, the global compilation problem is to find the optimal time allocation to all the nodes of the tree that maximizes the quality of the output of the root node:

$$Q_e^{\mathrm{G}}(t) = arg\max_{t_{i,j}} Q_{0,0}(\cdot), \qquad \sum_{0\leqslant i\leqslant n}\sum_{0\leqslant j\leqslant 2^i-1} t_{i,j} = t, \tag{14}$$

where $Q_{0,0}(\cdot)$ denotes the result of replacing (in the expression e) every function by its conditional performance profile and every input variable by its quality.

We define a local compilation scheme for e by induction on its structure. For a leaf node, the locally compiled performance profile is the conditional performance profile associated with that node:

$$Q_{n,j}^{\mathrm{L}}(t) = Q_{n,j}(q_{n,j,1}, q_{n,j,2}, t), \quad 0 \leqslant j \leqslant 2^n - 1, \tag{15}$$

where $q_{n,j,1}$ and $q_{n,j,2}$ are the qualities of the two inputs of the particular function. For each internal node, the locally compiled performance profile is defined using the performance profiles of its immediate inputs:

$$Q_{i,j}^{\mathrm{L}}(t) = arg\max_{t_1,t_2}\{Q_{i,j}(Q_{i+1,2j}^{\mathrm{L}}(t_1), Q_{i+1,2j+1}^{\mathrm{L}}(t_2), t - t_1 - t_2)\}. \tag{16}$$

Finally, the performance profile of e is denoted by the following expression:

$$Q_e^{\mathrm{L}}(t) = Q_{0,0}^{\mathrm{L}}(t). \tag{17}$$

Note that the external input quality was deliberately omitted in this notation since we focus on the result of local compilation for any *given* input quality. We are now ready to prove the following result:

Theorem 4.6 (Optimality of local compilation of functional expressions). *Let e be a functional expression of an arbitrary depth n whose conditional performance profiles satisfy the input monotonicity assumption, then for any input and total time allocation t:*

$$Q_e^{\mathrm{L}}(t) = Q_e^{\mathrm{G}}(t).$$

Proof. By induction on the depth of the tree. For trees of depth 1 the claim is trivially true because both compilation schemes solve the same optimization problem. Suppose that the claim is true for trees of depth $n-1$ or less. Let e be an expression of depth n, and let $t_{i,j}$ be the allocations to $f_{i,j}$ based on global compilation and resulting in a global optimum. Let t_l and t_r be respectively the total allocation to the left and right subtrees of the root node:

$$t_l = \sum_{i=1}^{n} \sum_{j=0}^{2^{i-1}-1} t_{i,j}, \tag{18}$$

$$t_r = \sum_{i=1}^{n} \sum_{j=2^{i-1}}^{2^i-1} t_{i,j}, \tag{19}$$

$$t = t_l + t_r + t_{0,0}. \tag{20}$$

Then:

$$Q_e^{\mathrm{G}}(t) =$$

By definition and input monotonicity

$$= Q_{0,0}(Q_{1,0}^{\mathrm{G}}(t_l), Q_{1,1}^{\mathrm{G}}(t_r), t_{0,0}) \tag{21}$$

By the induction hypothesis

$$= Q_{0,0}(Q_{1,0}^{\mathrm{L}}(t_l), Q_{1,1}^{\mathrm{L}}(t_r), t_{0,0}) \tag{22}$$

By definition of local compilation

$$\leqslant Q_{0,0}^{\mathrm{L}}(t) \tag{23}$$

By definition

$$= Q_e^{\mathrm{L}}(t) \tag{24}$$

By definition of global compilation

$$\leqslant Q_e^{\mathrm{G}}(t). \tag{25}$$

Hence $Q_e^{\mathrm{L}}(t) = Q_e^{\mathrm{G}}(t)$. □

Since local compilation yields optimal results, it is useful to determine the conditions under which it can reduce the complexity of the compilation problem. Obviously, if each function can take any number of arguments, we cannot guarantee any reduction in complexity. With unbounded number of inputs the depth of the corresponding tree may be one in which case local and global compilation solve the same problem. Hence, we will examine the complexity of local compilation under the *bounded degree assumption* that each node of the tree has a bounded degree. In other words, we assume that the number of inputs to each function is bounded. This assumption only reinforces the principle of modularity that has been long recognized in the development of complex systems.

Given a functional expression of size n and discrete performance profiles with maximal run-time t, we have the following result:

Theorem 4.7. *The tree-structured GCFE problem is polynomial in nt under the input monotonicity and bounded degree assumptions.*

Proof. Since local compilation guarantees optimality under input monotonicity and since local compilation needs to be repeated $O(n)$ times, we only need to show that local compilation of a single node is polynomial in t. This is trivially true under the bounded degree assumption. In particular, if the degree of each node is bounded by k, then the complexity of local compilation is $O(nt^k)$. Unless otherwise mentioned, we will assume in this section that $k = 2$ and that the complexity of local compilation is $O(nt^2)$. □

Note that there is no contradiction between this result and the NP-completeness of the KNAPSACK problem. Both the input monotonicity and the bounded degree assumptions are met by the reduction of Theorem 4.4. However, this does not imply that the KNAPSACK problem is polynomial in n. It does imply that the problem is polynomial in nt (where t represents the maximal element size), and this is already known. The dependency of the algorithm complexity on t is not a problem in our domain for several reasons. First, the range of possible run-times in a real-time system is normally bounded by some constant. Second, the fixed tabular representation of performance profiles allows us to limit the value of t by selecting the "appropriate" time unit for the application. When the performance profile size is bounded by a constant, local compilation can be performed in constant time at each node, and the complexity of the entire process becomes $O(n)$—linear in the program size.

In terms of space requirements, even though local compilation requires $O(n)$ separate performance profiles (one for each internal node of the tree), its total space requirement is only a constant factor more than the space requirement of global compilation. This is due the fact that a globally compiled performance profile must specify the allocation to *each* node of the tree while a locally compiled performance profile needs to specify only the allocation to the immediate successors of each node and to the node itself. To summarize, local compilation has the same space complexity as global compilation but it reduces the time complexity of the optimization problem from exponential to polynomial in nt.

4.3. Additional compositional operators

The family of functional expressions can be enriched with a large set of standard compositional operators. The optimality of local compilation remains valid as long as each operator, ϕ, satisfies two requirements: (1) the operator produces a result whose quality depends on the qualities of its inputs and on the amount of time allocated to the evaluation of the operator itself, t_ϕ; and (2) the conditional performance profile of the operator exhibits input monotonicity. Many useful operators satisfy these requirements. In many cases the evaluation time of such operators is a small constant time and their conditional performance profiles are represented as step functions.

For example, consider the operator *oneof*:

$$F(x) = oneof(M_1(x), \ldots, M_n(x)).$$

The output of *oneof* is the result of its single component, M_i, with the highest quality and its quality is the quality of that component. Suppose that each component, M_i, is an anytime algorithm whose performance profile is Q_i. The conditional performance profile of *oneof* is:

$$Q_{oneof}(q_1, \ldots, q_n, t) = \begin{cases} \max(q_1, \ldots, q_n), & \text{if } t > t_{oneof}, \\ 0, & \text{otherwise.} \end{cases} \tag{26}$$

This models a situation in which several alternative methods can be used to solve the same problem. For example, suppose that one needs to transport n identical packages using a certain container. The components of *oneof* might be several alternative bin packing algorithms where the quality of each algorithm is measured by the portion of the container's volume filled with packages. Obviously, the maximal volume that can be transported is proportional to the maximal quality among all the individual bin packing algorithms. Additional examples of such compositional operators appear in [34].

4.4. Repeated subexpressions

Local compilation does not produce good results when applied to functional expressions with repeated subexpressions. Using the tree representation, a repeated subexpression corresponds to a repeated subtree. The problem with local compilation is that it allocates computation time to *all* the nodes of the tree while time should be allocated only once to evaluate all the copies of a repeated subexpression. For example, consider the functional expression that appears in Fig. 13.

$$F(x) = E(D(B(A(x)), C(A(x)))).$$

The subexpression $A(x)$ appears twice and an efficient compiler should not allocate time to both copies. This means, however, that the allocation of time to $A(x)$ cannot be done locally, since it affects the output qualities of both C and D.

In this section, we present three time allocation methods that deal with general functional expressions:

- HILL-CLIMBING-ALLOCATION finds a solution to the global compilation problem directly, but does not guarantee global optimality.
- CONDITIONING-ALLOCATION tries all possible allocations to the repeated subexpressions, then applies local compilation to the resulting trees (note the analogy to conditioning methods in belief network evaluation [25]).
- TRADING-ALLOCATION begins with the allocation determined by local compilation, and then trades time among components so that only one copy of each repeated subexpression ends up with a nonzero allocation.

All three methods where developed using the discrete tabular representation of performance profiles. The complexity and optimality of the three methods are discussed below. For each algorithm we will compute the complexity of calculating *each* entry of

```
HILL-CLIMBING-ALLOCATION
1   for each Q_in ∈ [Q_L .. Q_U] do
2     for each T ∈ [T_L .. T_U] do
3       s ← INITIAL-RESOLUTION(T)
4       t_i ← T/n   ∀i: 1 ⩽ i ⩽ n
5       repeat
6         while ∃i, j such that
            E(Q_out(Q_in, t_1, ..., t_i − s, ..., t_j + s, ..., t_n)) >
            E(Q_out(Q_in, t_1, ..., t_n))
7           let i, j be the ones that maximize expected quality
8           t_i ← t_i − s
9           t_j ← t_j + s
10        s ← s/2
11      until s < ε
12      T[Q_in, T] ← (t_1, ..., t_n)
```

Fig. 15. Time allocation using a hill-climbing search.

the table representing the performance profile (that is, the complexity of calculating the optimal allocation to the components for any particular input quality and total run-time).

4.4.1. Time allocation using a hill-climbing search

This time allocation algorithm uses the DAG representation of functional expressions. For each particular time allocation to the components of a DAG, the quality of the output can be computed using the conditional performance profiles of the components. This computation can be performed in linear time in the size of the graph. While the search space of all possible time allocations has exponential size, an efficient hill-climbing search procedure can be constructed by limiting the search space.

The time allocation algorithm, shown in Fig. 15, starts with an equal amount of time allocated to each component of the DAG. Then it considers trading s time units between two modules so as to increase the expected quality of the output. As long as it can improve the expected quality, it trades s time units between the two modules that have maximal effect on output quality. When no such improvement is possible with the current value of s, it divides s by 2 until s reaches a certain minimal value, ϵ. At that point, it reaches a local maximum and returns the best time allocation it found. As with any hill-climbing algorithm, it suffers from the problem of converging on a local maximum. An analysis of the algorithm shows that simple properties of the conditional performance profiles of the components, such as monotonicity, are not sufficient to guarantee global optimality.

Complexity

Let κ be the size of the functional expression (i.e. the number of nodes in the corresponding DAG), and let $\tau = T_{max}/\epsilon$ be the maximal number of discrete time units to be allocated. The complexity of the algorithm is then $O(\kappa^3 \log \tau)$. This is due to

```
CONDITIONING-ALLOCATION
1   for each Q_in ∈ [Q_L .. Q_U] do
2     for each T ∈ [T_L .. T_U] do
3       Q_max ← 0
4       r_max ← 0
5       for r ← 0 to T step ε
6         t ← T − r
7         ADJUST-PP(r)
8         APPLY-LOCAL-COMPILATION(e, t)
9         Q ← Q_out(Q_in, (r | t_1, ..., t_{n−m}))
10        if Q > Q_max then do
11          Q_max ← Q
12          A_opt ← (r | t_1, ..., t_{n−m})
13      T[Q_in, T] ← A_opt
```

Fig. 16. Time allocation with pre-determined time to repeated subexpressions.

the fact that for each search resolution s, the algorithm needs to find the optimal pair of modules for trading time. This is done in $O(\kappa^2)$ by considering every possible pair. This step repeats only a constant number of times. Finding the expected quality of the output is performed in $O(\kappa)$ and the number of time resolution steps is $O(\log \tau)$.

4.4.2. Pre-determined allocation to repeated subexpressions

The second method, CONDITIONING-ALLOCATION, is based on fixing the allocation to each repeated subexpression before computing the allocation to the other components. The allocation to the other components is determined based on standard local compilation. Time allocation is made only once to all the copies of each repeated subexpression. Once that allocation is decided, the complete expression is treated as a tree rather than a DAG and the efficient local compilation scheme is used.

Let e be a functional expression of size n. Assume that e has only one repeated subexpression e' that appears $m > 1$ times in e. The copies of e' are denoted by $e'_1, \ldots, e'_m$. Let $(r \mid t_1, \ldots, t_{n-m}))$ represent allocation of r time units to $e'_1, \ldots, e'_m$ and $t_1, \ldots, t_{n-m}$ to the remaining $n-m$ modules. Fig. 16 shows the time allocation algorithm. Its central idea is to *reserve* a certain amount of time r, out of the total allocation t, for evaluating a single copy of the repeated subexpression e'. All the other copies "enjoy for free" the result of this evaluation. The fact that r time units are reserved for e' is communicated to the local compilation process by adjusting the performance profile of e'. The new performance profile is a step function that returns quality $Q_{e'}(r)$ at zero time and provides no further improvement of quality. Since no improvement of quality is possible, an optimal schedule would not allocate time to any of the copies and hence standard local compilation is guaranteed to allocate the remaining time optimally to the *other* components. The algorithm performs a search to find the best pre-determined reserved time r for which the output quality is maximal.

If the conditional performance profiles of all the components of e satisfy the input monotonicity assumption, then any optimal schedule has the following property:

Lemma 4.8. *Any optimal schedule for the evaluation of e allocates time to a single copy of e'.*

Proof. Suppose that there is an optimal schedule in which more than one copy of e' is evaluated. Let $r_1, \ldots, r_m$ be the allocations to the m copies, and let $r = \sum r_i$. By the monotonicity of the performance profile of e', the quality achieved by allocating r time units to a single copy is greater than any of the qualities achieved with allocations $r_1, \ldots, r_m$. Hence, by substituting the result of that single copy for all the copies without changing the allocation to the other components, and by the monotonicity of the conditional performance profiles, it is apparent that the output quality would increase. This contradicts the optimality of the original schedule. Therefore, time must be allocated to a single copy only. □

CONDITIONING-ALLOCATION can be viewed as a two phase optimization process. Its first phase determines the optimal r and its second phase finds the optimal allocation to the other components. Having established the fact that any optimal schedule must activate e' only once, we can conclude the global optimality of this method:

Theorem 4.9 (Optimality of CONDITIONING-ALLOCATION). *Let e be a functional expression with a single repeated subexpression e', then* CONDITIONING-ALLOCATION *returns a globally optimal schedule for evaluating e.*

Proof. An immediate result of Lemma 4.8 and the optimality of local compilation. □

Complexity

Again, let κ be the size of the functional expression and let $\tau = T_{\max}/\epsilon$ be the maximal number of time units to be allocated. The complexity of the algorithm is then $O(\kappa\tau^3)$. This is due to the fact that the complexity of the search for the optimal value of r is $O(\tau)$ and the most complicated step inside the loop is local compilation with complexity $O(\kappa\tau^2)$.

To extend this method to work with p different repeated subexpressions, the algorithm must consider any possible pre-determined allocation to (single copies of) each repeated subexpression. The complexity of this step is $O(\tau^p)$ when $p \ll \tau$. The overall complexity becomes $O(\kappa\tau^{(p+2)})$.

4.4.3. Learning the allocation to repeated subexpressions

The third method, TRADING-ALLOCATION, is based on learning the allocation to repeated subexpressions through standard local compilation. To be able to apply local compilation, the algorithm first ignores the repetition of subexpressions and uses standard local compilation. Then it applies a series of performance profile adjustments followed by local compilation. The process converges on a single allocation to each repeated subexpression.

```
TRADING-ALLOCATION
1   for each Q_in ∈ [Q_L .. Q_U] do
2     for each T ∈ [T_L .. T_U] do
3       r ← 0
4       repeat
5         t ← T − r
6         SHIFT-PP(r)
7         APPLY-LOCAL-COMPILATION(e, t)
8         Let r_1, ..., r_m be the allocations to e_1, ..., e_m
9         r ← r + max{r_i}
10      until Σ r_i = 0
11      T[Q_in, T] ← (r | t_1, ..., t_n)
```

Fig. 17. Learning the allocation to repeated subexpressions.

Again, let e be a functional expression. As with CONDITIONING-ALLOCATION, we consider first the case where e has only one repeated subexpression e' with copies $e'_1, \ldots, e'_m$. Fig. 17 shows the time allocation algorithm. It learns the allocation r to a single copy of e'. Starting with $r = 0$, the algorithm repeatedly increases r until local compilation allocates no additional time to the copies of e'. In each iteration, the current value of r is used to determine how much time to reserve for evaluating e'. The fact that r time units are reserved for e' is communicated to the local compilation process by adjusting the performance profile of e'. The time origin of the performance profile is shifted r units to the right. Standard local compilation is then applied and the optimal allocation to *all* the components is computed. Suppose that, based on the adjusted performance profile, the allocations to the m copies of e' are $r_1, \ldots, r_m$. Then, the maximal allocation among those is used to increase the value of r. This process is repeated until no additional time is allocated to any of the copies beyond the reserved time allocation r. This time allocation algorithm does not guarantee global optimality [34].

Complexity

Using the same notation as above, the complexity of the algorithm is $O(\kappa\tau^3)$. This is due to the fact that the complexity of the search for r is $O(\tau)$ (since r may be incremented by 1 unit of time in each iteration). The most complicated step inside the loop is local compilation with complexity $O(\kappa\tau^2)$. Note that in practice the convergence of the search for r is much faster than $O(\tau)$.

The extension to multiple repeated expressions is straightforward. The algorithm needs to maintain a sequence of reserved allocations for each repeated subexpression. The rest of the algorithm remains the same. The advantage of TRADING-ALLOCATION is that its complexity grows only linearly with the number of repeated subexpressions, p. This is due to the fact that a single loop is used to update *all* the reserved allocations to repeated subexpressions and the worst case complexity of that loop is only $O(p\tau)$. Hence the overall complexity in the general case is $O(\kappa p\tau^3)$.

Summary

We have examined three time allocation algorithms designed to cope with the difficulty of compiling general functional expressions. HILL-CLIMBING-ALLOCATION has a complexity $O(\kappa^3 \log \tau)$ and finds only local optimum. CONDITIONING-ALLOCATION has complexity $O(\kappa\tau^{(p+2)})$ and TRADING-ALLOCATION $O(\kappa p\tau^3)$. When $\kappa \ll \tau$ the first algorithm is the most efficient one. The second method guarantees optimality, but its complexity grows exponentially with the number of repeated subexpressions. To address this problem, the last method can be used. Its complexity grows only linearly with the number of repeated subexpressions but it does not guarantee global optimality. By using local compilation to determine the allocation to the rest of the components, TRADING-ALLOCATION is more likely to converge on the global optimum than HILL-CLIMBING-ALLOCATION.

5. Conclusion

This paper examines the possibility of extending the advantages of anytime algorithms to the construction of complex real-time systems. The first results on this vital problem show that a modular composition of anytime algorithms can be implemented efficiently. In particular, we show that:

(1) The performance profile of a composite system can be derived automatically and efficiently by off-line compilation techniques. The compilation process optimizes the overall quality of the system as a *contract* algorithm.

(2) The resulting system can be made interruptible with only a small, constant penalty.

(3) Our approach separates two central aspects of system development, namely the construction of the performance components and the optimization of performance. In real-time system construction this separation isolates each module from the time constraints that it must satisfy. As a result, our compilation mechanism simplifies the design of real-time systems and allows for modularity and abstraction to be applied.

(4) The resulting real-time system is machine-independent in the sense that it can adapt its internal time allocation to the available computational resources.

The main contribution of the paper includes: (1) formalizing the compilation problem and solving it for the case of functional composition; (2) making the interruptible/contract distinction that facilitates a two-step solution to the compilation problem; and (3) formalizing the notion of conditional performance profiles that allow us to solve a large part of the problem off-line.

Further work in this area is currently aimed at: (1) developing larger applications to further evaluate the components of the model; (2) extending the scope of compilation by studying the compilation of additional programming structures; (3) extending the scope of anytime algorithms to include anytime sensing and anytime action; and (4) building a programming environment to support anytime algorithm development. Our ultimate goal is to construct robust real-time systems in which perception, deliberation and action are governed by a collection of anytime algorithms.

Acknowledgements

The authors would like to thank the members of Shlomo Zilberstein's dissertation committee—Alice Agogino, Tom Dean and Susan Graham. Tom Dean in particular inspired an initial interest in anytime algorithms and has continued to provide important insights and comments. Thanks also to the members of the RUGS seminar—Francesca Barrientos, Othar Hansson, Tim Huang, Andrew Mayer, Ron Musick, Gary Ogasawara and Ron Parr. This research was partially supported by the National Science Foundation under grants IRI-8903146 and IRI-9058427 and by the Malcolm R. Stacey Fellowship to Shlomo Zilberstein.

References

[1] M. Boddy, Anytime problem solving using dynamic programming, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 738–743.

[2] M. Boddy and T.L. Dean, Solving time-dependent planning problems, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 979–984.

[3] P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor and D. Freeman, Autoclass: a Bayesian classification system, in: *Proceedings Fifth International Conference on Machine Learning*, Ann arbor, MI (1988).

[4] B. D'Ambrosio, Resource bounded agents in an uncertain world, in: *Working Notes of the IJCAI-89 Workshop on Real-Time Artificial Intelligence Problems*, Detroit, MI (1989).

[5] R. Davis, Meta-rules: reasoning about control, *Artif. Intell.* **15** (1980) 179–222.

[6] T.L. Dean, Intractability and time-dependent planning, in: M.P. Georgeff and A.L. Lansky, eds., *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, Los Altos, CA (Morgan Kaufmann, 1987).

[7] T.L. Dean and M. Boddy, An analysis of time-dependent planning, in: *Proceedings AAAI-88*, St. Paul, MN (1988) 49–54.

[8] J. Doyle, Rationality and its roles in reasoning, in: *Proceedings AAAI-89*, Boston, MA (1990) 1093–1100.

[9] C. Elkan, Incremental, approximate planning: abductive default reasoning, in: *Working Notes of the AAAI Spring Symposium on Planning in Uncertain Environments*, Palo Alto, CA (1990).

[10] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, San Francisco, CA, 1979).

[11] A. Garvey and V. Lesser, Design-to-time real-time scheduling, in: *IEEE Trans. Syst. Man Cybern.* **23** (6) (1993).

[12] M.R. Genesereth, An overview of metalevel architectures, in: *Proceedings AAAI-83*, Washington, DC (1983) 119–123.

[13] J.A. Hendler, Real-time planning, in: *Working Notes of the AAAI Spring Symposium on Planning and Search*, Stanford, CA (1989).

[14] E.J. Horvitz, Reasoning about beliefs and actions under computational resource constraints, in: *Proceedings 1987 Workshop on Uncertainty in Artificial Intelligence*, Seattle, WA (1987).

[15] E.J. Horvitz and J.S. Breese, Ideal partition of resources for metareasoning, Technical Report KSL-90-26, Stanford Knowledge Systems Laboratory, Stanford, CA (1990).

[16] E.J. Horvitz, H.J. Suermondt and G.F. Cooper, Bounded conditioning: flexible inference for decision under scarce resources, in: *Proceedings 1989 Workshop on Uncertainty in Artificial Intelligence*, Windsor, Ont. (1989) 182–193.

[17] R.M. Karp, An introduction to randomized algorithms, Technical Report TR-90-024, International Computer Science Institute, Berkeley, CA (1990).

[18] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, *Artif. Intell.* **27** (1985) 97–109.

[19] R.E. Korf, Real-time heuristic search, *Artif. Intell.* **42** (3) (1990) 189–212.

[20] E.L. Lawler et al., eds., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization* (Wiley, New York, 1987).
[21] V. Lesser, J. Pavlin and E. Durfee, Approximate processing in real-time problem-solving, *AI Mag.* **9** (1) (1988) 49–61.
[22] S. Lin and B.W. Kernighan, An effective heuristic algorithm for the Traveling Salesman Problem, *Oper. Res.* **21** (1973) 498–516.
[23] T. Lozano-Pérez and R.A. Brooks, An approach to automatic robot programming, in: M.S. Pickett and J.W. Boyse, eds., *Solid Modeling by Computers* (Plenum, New York, 1984) 293–327.
[24] R.S. Michalski and P.H. Winston, Variable precision logic, *Artif. Intell.* **29** (2) (1986) 121–146.
[25] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference* (Morgan Kaufmann, Los Altos, CA, 1988).
[26] A. Pos, Time-constrained model-based diagnosis, Master Thesis, Department of Computer Science, University of Twente, Enschede, Netherlands (1993).
[27] S.J. Russell, D. Subramanian and R. Parr, Provably bounded optimal agents, in: *Proceedings IJCAI-93*, Chambery, France (1993) 338–344.
[28] S.J. Russell and E.H. Wefald, Principles of metareasoning, in: R.J. Brachman et al., eds., *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning* (Morgan Kaufmann, San Mateo, CA, 1989).
[29] S.J. Russell and E.H. Wefald, *Do the Right Thing: Studies in limited rationality*, (MIT Press, Cambridge, MA, 1991).
[30] S.J. Russell and S. Zilberstein, Composing real-time systems, in: *Proceedings IJCAI-91*, Sydney, Australia (1991) 212–217.
[31] H.A. Simon, *Models of Bounded Rationality, Vol. 2* (MIT Press, Cambridge, MA, 1982).
[32] S.V. Vrbsky and J.W.S. Liu, Producing monotonically improving approximate answers to database queries, in: *Proceedings IEEE Workshop on Imprecise and Approximate Computation*, Phoenix, AZ (1992) 72–76.
[33] L.A. Zadeh, Fuzzy logic and approximate reasoning, *Synthese* **30** (1975) 407–428.
[34] S. Zilberstein, Operational rationality through compilation of anytime algorithms, Ph.D. Dissertation, Computer Science Division, University of California, Berkeley, CA (1993).
[35] S. Zilberstein and S.J. Russell, Efficient resource-bounded reasoning in AT-RALPH, in: *Proceedings First International Conference on AI Planning Systems*, College Park, MD (1992) 260–266.
[36] S. Zilberstein and S.J. Russell, Constructing utility-driven real-time systems using anytime algorithms, in: *Proceedings IEEE Workshop on Imprecise and Approximate Computation*, Phoenix, AZ (1992) 6–10.
[37] S. Zilberstein and S.J. Russell, Anytime sensing, planning and action: A practical model for robot control, in: *Proceedings IJCAI-93*, Chambery, France (1993) 1402–1407.