ornl

ORNL/TM-9987
CESAR-86/09

OAK RIDGE
NATIONAL
LABORATORY

MARTIN MARIETTA

ROSES, A Robot Operating System Expert Scheduler: Methodological Framework

E. C. Halbert
J. Barhen
P. C. Chen

OPERATED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

ORNL/TM--9987

DE90 014782

Engineering Physics and Mathematics Division

ROSES, A ROBOT OPERATING SYSTEM EXPERT SCHEDULER: METHODOLOGICAL FRAMEWORK

E. C. Halbert
J. Barhen*
P. C. Chen†

*Currently at Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA
†Currently at Sandia National Laboratory, Albuquerque, NM

Date of Issue: August 1990

Prepared by the
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831
operated by
MARTIN MARIETTA ENERGY SYSTEMS, INC.
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-84OR21400

TABLE OF CONTENTS

ABSTRACT

The optimal scheduling of tasks among which complex interrelationships (such as precedence constraints) may exist is essential for driving the new generation of concurrent supercomputers to their utmost performance. To address this need, the project ROSES (Robot Operating System Expert Scheduler) has been initiated at the Oak Ridge National Laboratory's Center for Engineering Systems Advanced Research. The project, its method for optimizing schedules, and its implementing computer code are each called ROSES. The problem of finding optimum schedules is explosive in complexity (i.e., NP-complete). By combining heuristic techniques, graph-theoretic algorithms, and sophisticated data structures, ROSES achieves near-optimal solutions in a highly efficient manner both with respect to computer time and memory-space. In this report, the description of the methodology is followed by an overview of the ROSES computer code, including detailed outlines of selected algorithms. In addition, this report describes an application of ROSES to schedule inverse dynamics computations for a robot manipulator.

EXECUTIVE SUMMARY

Some of the most challenging computational problems being posed today concern intelligent autonomous systems, such as autonomous robots. These systems must perform highly complex computations very rapidly. To meet the demands, concurrent computation seems necessary.

ROSES is an optimization procedure that searches to find rapid schedules for concurrent-computation performance of precedence-constrained tasks.

Some of the terminology warrants comment. *Concurrent computations* refers to the use of an ensemble of processors that work simultaneously, though not necessarily in lockstep fashion, toward completing a large computational job. To prepare for concurrent computation, the large job is broken up into atomic *computational tasks*, each to be performed wholly on a single processor of the ensemble. A *schedule* is a plan that partitions the computational tasks among concurrent processors, and indicates when each of these tasks is to begin. A *rapid* schedule is one for which the wall-clock time is short from start to scheduled finish of the multitask job. An *optimum* schedule is most rapid (it finishes in minimum time). Usually there are *precedence constraints* to satisfy. For example, it may be that computational task i cannot start until computational tasks a and b are completed, because task i needs results from the completed tasks a and b. In that case: unless tasks i and a and b are all assigned to the same processor, there will be a need for messages to pass results from a and/or b to i.

Determining the optimum constraint-satisfying schedule is an NP-complete problem (i.e., is explosive in complexity). ROSES finds near-optimum constraint-satisfying schedules quickly.

ROSES can generate near-optimum schedules for multitask jobs having this property: The time needed for message-passing is small compared with the time needed to process computational tasks (no matter what the partition of tasks among processors).

This report describes the ROSES search-scheme in terms of: two different graphs (a search tree and a task graph), heuristics based largely on critical-path considerations (plus other time bounds), and various data structures. All these combine to make the ROSES search-process efficient.

The schedule emergent from a ROSES search is called a ROSES *search-output schedule.* It always fully satisfies all the precedence constraints among computational tasks. However, because of simplifying approximations (e.g., in the search-scheme's model for message-passing), the ROSES search-output schedule will generally not be precisely executable on a real multiprocessor. What will be executable is a well-defined compromise schedule — a schedule which retains certain selected features of the ROSES search-output schedule, but replaces others (e.g., replaces all model message-passing times with real message-passing times). The compromise-schedule fully satisfies all the precedence constraints. It is easy to arrange for the multiprocessor to execute this compromise-schedule.

ROSES has been applied to schedule the solution of dynamics equations for a robot arm. The ROSES search-output results were compared with results obtained earlier

by other workers using different methods. In several ways, the results from ROSES were superior to the earlier results.

I. INTRODUCTION

Some of the most challenging computational problems facing scientists and engineers today arise within the framework of intelligent autonomous systems. Such problems range from robots operating in unstructured hazardous environments where explosives, toxic chemicals, or radioactivity may be present, to the development of battle management paradigms for the Strategic Defense Initiative. To enable a robotic system to work effectively in real time in an unstructured environment, one needs to solve repeatedly a variety of highly complex mathematical problems such as on-line planning, vision, sensor fusion, navigation, manipulator dynamics and control. Similarly, a "Star-Wars" defense system would need to respond to an offensive strike by coordinating perhaps millions of separate actions on a schedule timed in milliseconds. The computational requirements of these problems fall into the "supercomputer" class, but ultimately we need to solve them "onboard" the autonomous robot or space system. The only realistic option is VLSI-based concurrent computation.

Concurrent computation[1] refers to the use of an ensemble of small computers that work in parallel, but not necessarily in lockstep fashion, on parts of a complex problem. Our interest is in concurrent processors which coordinate their activities almost entirely by sending messages to each other[2] (to avoid many of the difficulties associated with attempts to scale-up shared memory systems to a large number of processors). The major technological drive behind concurrent computation appears to be very large scale integration (VLSI).[3] The basic trend is to use state-of-the-art VLSI to integrate an entire processing system on a single chip[4-5] including communication links, memory interface, 32-bit processors and even 64-bit IEEE floating point,[5] resulting in smaller and cheaper individual processors comparable in performance to their larger capability and more expensive predecessors. Such advanced single-chip processors are being combined into concurrent systems of great power and versatility. Methods must be developed for properly applying this power and versatility to the complex problems needing solution by intelligent autonomous systems.

One of the first and most important steps in attempting to solve a problem in a concurrent computation ensemble is decomposing the problem into a set of tasks (or "processes") each to be performed wholly by one processor of the ensemble. When a problem is re-expressed by decomposing it, the re-expression often involves precedence constraints among the tasks. The distributed nature of the computational system translates these precedence constraints into message-passing requirements. A plan which partitions the task set among the processors, and which also stipulates when each task should be performed, is called a "schedule". The optimal scheduling of tasks, among which complex interrelationships (such as precedence constraints) may exist, is essential for driving the new generation of concurrent supercomputers to their utmost performance. To address this need, the project ROSES (Robot Operating System Expert Scheduler) has been initiated at the Oak Ridge National Laboratory's Center for Engineering Systems Advanced Research. The project, its method for optimizing schedules, and its implementing computer are each called ROSES.

The ROSES methodology and computer code are due to two of this report's authors. The methodology was conceived by J. Barhen. The code was written by J. Barhen and P. C. Chen.

This report focuses on the methodological framework of ROSES. The problem of finding optimal schedules is NP-complete. It is computationally intractable[6] for more than two processors and task sets of large dimensionality. ROSES has been designed to achieve near-optimal schedules by combining heuristic techniques and graph-theoretic algorithms to control time complexity, with sophisticated data structures to handle efficiently space complexity.

The major part of ROSES proceeds by search to generate a "ROSES search-output schedule." The course of the search, its emergent schedule, and ROSES's estimated time for concurrently executing that schedule, are all influenced by input data describing the computational tasks, message requirements, and processors. The input data include, for example: (1) each computational task's algorithmic parameters, such as the number of floating point multiplications in the task; and (2) processor system performance parameters, such as the time for one floating point multiplication. Thus, ROSES can be used to estimate optimal concurrent-execution times for various problem decompositions and for various multiprocessors — all these estimates being obtainable without any actual experimental runs on a concurrent multiprocessor.

After an acceptable problem decomposition and ROSES search-output have been determined, actual execution on a multiprocessor is in order. To facilitate such execution, we invoke an optional, concluding part of the ROSES computer code. This optional part generates "run-time" control data to be fed to the multiprocessor along with routines corresponding to all the individual computational tasks that ROSES has scheduled. (Also to be fed are an appropriate host program, and ordinary input data to complete the description of the multi-task problem.) The run-time control data will inform the multiprocessor how to execute the multi-task job in accord with selected features of the ROSES search-output schedule. The schedule that will be executed will be in perfect compliance with the precedence constraints.

This report is organized as follows. Section II provides a short background on multiprocessor scheduling. The methodological framework for ROSES schedule-optimization is described in Section III. This is followed, in Section IV, by an overview of the schedule-optimization part of the ROSES computer code. Detailed outlines of selected routines are included. Section V discusses an application of ROSES: scheduling the solution of inverse dynamics equations for a robot manipulator. The Section V discussion includes some further description of the "run-time" data generated by ROSES. Section VI contains brief comments on future research directions.

II. BACKGROUND

Multiprocessor scheduling has been studied extensively over the last twenty years, and excellent reviews can be found in the literature.[7-10] Most earlier work for processor scheduling has been concerned with deterministic models[7-9] in which the task execution times are assumed to be fixed and known in advance. Stochastic models, in which task execution times are random variables and the computer system is modeled via queuing networks, will not be discussed here, but the interested reader is referred to the articles of Robinson[10] and Rosberg[11] as a starting point for further details. Major complications in deterministic scheduling arise when the number of tasks in a particular algorithm exceeds the number of available processors, and/or when the topology of the task graph (as determined by the precedence constraints) is not expressible in a way similar to the interconnection topology of the computation ensemble. Furthermore, optimal schedules are in general extremely difficult if not impossible to obtain, since for an arbitrary number of processors, unequal task processing times, and non-trivial precedence constraints, the problem is NP-complete.[6]

In the past, studies of multiprocessor scheduling have often been abstract, since few appropriate machines existed. Among the many outstanding contributions made, in this area of abstract studies, special credit goes to the seminal paper of Ramamoorthy, Chandy and Gonzalez (RCG).[12] They present algorithms and heuristics to obtain the minimum number of processors such that given a set of computational tasks and their partial ordering, the overall execution time would be minimized. RCG also address the more general problem of scheduling precedence-constrained tasks on an arbitrary number of processors. In particular, they point out that for multiprocessor scheduling problems involving tasks with unequal processing times, it is sometimes optimal to keep a processor idle even when there are tasks that could be processed immediately. In a similar vein, Fernandez and Bussel have derived[13] sharp lower bounds on the number of processors and on the minimum time required for optimal schedules. Bounds on the performance guarantees of scheduling algorithms have also been investigated by Garey, Graham and Johnson,[14] and more recently by Sarin and Elmaghrabi.[15]

Another important research area is concerned with the problem of scheduling T tasks, where each task i has a release time r_i and a deadline time d_i. For a variable number of processors, if tasks are related by a partial order (i.e., by a relation that is reflexive and transitive) and are allowed to have different processing times t_i, then the problem is known to be strongly NP-complete.[6] This is still the case even if the tasks are to be scheduled on a single processor.[16] Ullman has shown[17] that the problem remains NP-complete for unit-time tasks, even with all release times set to zero and a single deadline. However, Simons and Sipser[18] were able to prove that if no partial ordering applies, a polynomial time algorithm exists, assuming again unit-time tasks but with unconstrained release times and deadlines. They have also investigated how small perturbations of the basic assumptions underlying a scheduling problem impact its transition from NP to P-completion and vice versa.

Monma's work focuses on scheduling T equal-length tasks on N identical parallel processors, subject to "in-tree" precedence constraints.[21] This corresponds to a situation in which all tasks have at most one immediate successor. Monma shows that for tasks with different completion deadlines an $O[T]$ linear-time algorithm can be derived.

The effect of processor memory size on the scheduling problem has been considered by Lai and Sahni.[22] They show that no multiprocessor system that contains at least one processor with memory size smaller than at least two other processors can be scheduled "nearly on-line" to minimize the overall completion time.

Numerous other interesting algorithms have been proposed, including the use of quadratic programming[23] and linear execution orderings via nonlinear integer optimization.[24] A great deal of effort has also been devoted to load-balancing issues.[25] However, the applicability of most proposed approaches to real-life problems suffers from some of their underlying assumptions (e.g., "unit-time" task lengths, and/or few processors, and/or no precedence constraints).

With the motivation of newly developed concurrent computers, there has been an increased interest in formulating efficient multiprocessor scheduling algorithms.[26–29] Real-time systems in general, and robotic applications in particular, are the driving force behind these developments. Other promising avenues being explored involve the derivation of parallel scheduling algorithms.[30] Published results in this area are currently limited to SIMD architectures. Much work remains to be done, in particular for the more difficult MIMD, message-passing architectures.

Advanced autonomous robots, such as the HERMIES-II prototype currently being developed and tested at CESAR[31] or the Hexapod walking machine constructed by Ohio State University,[32] and other intelligence-targeted machines of the future, are generally composed of a variety of asynchronously controlled components. For a robot, these components may include manipulator arms, electro-optical sensors, sonars, navigation controllers, etc. In order to take advantage of the distributed nature of the associated robotic processes, it was envisioned[33] that a Robot Operating System (ROS) should be developed to provide a generalized framework for implementing machine intelligence in a real-time environment. ROSES (i.e., the ROS Expert Scheduler) is being developed in that context.

III. METHODOLOGICAL FRAMEWORK

This section focuses on the basic methodological framework which allows ROSES to find near-optimal solutions to the NP-complete precedence-constrained scheduling problem. Our intent is to show how ROSES combines heuristic techniques, graph-theoretic algorithms, and sophisticated data structures so as to make its search efficient. This section's statements about ROSES refer to the major, schedule-generation part of ROSES.

Concerning notation: In this report's variable-names, italic style and roman style are used interchangeably. So are hyphens and underlines. For example, *spt* and spt are identical in meaning; so are *TIME_to_GOAL* and TIME-to-GOAL.

III.1. STATEMENT OF THE PROBLEM

ROSES addresses problems of the following kind: Assume that one is given a set of N_e tasks $\{e|e = 1, ..., N_e\}$, each task having an expected time-length *L(e)*. An overall goal exists, and there are precedence constraints among the tasks and the goal. These precedence constraints are expressible in the form: "Tasks $e_a, \ldots, e_d$ must be completed before any of tasks $e_f, \ldots, e_i$ can begin; tasks $e_h, \ldots, e_m$ must be completed before any of tasks $e_r, \ldots, e_w$ can begin; ... tasks $e_s, \ldots, e_z$ must be completed before the goal is reached." In the foregoing statement, the subsets of predecessor tasks (that is, subsets like $e_a, \ldots, e_d$, $e_h, \ldots, e_m$, $e_s, \ldots, e_z$) may or may not be disjoint from each other; and similarly, the subsets of successor tasks may or may not be disjoint from each other. In any case, the cumulative requirement is that, in order to reach the goal, all the tasks must be completed — once each, in any sequence compatible with the precedence constraints. The tasks are to be performed on a set of N_p identical concurrent processors $\{p|p = 1, \ldots, N_p\}$. Each processor can handle only one task at a time, and task preemption is currently not allowed; i.e., once a task is started on a processor, that task is processed there without interruption until finished. The sooner the goal is reached, the better. Therefore the objective is to find an optimal or near-optimal schedule for reaching the goal rapidly.

Three "allowed approximations" deserve mention; they are used in translating real-world conditions to the above-described problem statement. These three approximations concern time-lengths, message-passing, and interprocessor relations. Each is listed and discussed separately.

1. Optimization is done assuming all time-lengths *L(e)* to be fixed, certain, and known in advance of task-performance. Zero-length tasks are allowed.

Though zero-length tasks do not directly consume scheduled time, they can of course imply precedence constraints. They can be useful for re-expressing a ROSES problem in terms of disjoint precedence requirements, as follows: Suppose that a task-scheduling problem is initially posed in terms of non-disjoint predecessor subsets and/or non-disjoint successor subsets. Then by postulating some extra zero-length tasks, it is always possible to pose an equivalent problem having only disjoint predecessor subsets and disjoint successor subsets. Such disjointness is convenient and is required for the main search algorithm of ROSES.

2. Communications between tasks are themselves treated as tasks. They are "message-passing tasks" as opposed to the others, which are called "computational tasks." The ROSES search algorithm treats all tasks alike. That is: during search ROSES assigns processors to tasks on the basis of the tasks' stated lengths and precedence constraints, with no other variables to indicate whether a task is message-passing or computational.

This modeling of messages is reasonable if message-passing times are small compared with computational-task times. For such cases, ROSES users may often decide to describe the message-passing tasks as having zero time-lengths.

3. Besides being identical in themselves, all processors are presently considered identical in their relationships to every other processor (and even reflexively, to themselves).

Thus, for example, all processors are assumed to be equidistant for message-passing (and at present there is no coded provision for avoiding a message-passing task, even if it connects two tasks assigned to be performed successively by the same processor). The assumption of equidistant processors may be viewed as a compromise, reasonable for general application to concurrent computers with a variety of special (but highly connected) configurations. In any case, the entire message-passing approximation still leaves ROSES in good competitive status with respect to other current scheduling systems.

Our expectation is that forthcoming improvements in hardware will greatly reduce message-passing times compared to the times of logical and arithmetic operatives on a single processor. Therefore, we expect that neglecting message-passing times will often be a very reasonable approximation in searches to optimize schedules.

III.2. SPECIFICATION OF SCHEDULES

As noted, the problem is to find an optimal or near-optimal schedule for reaching the goal rapidly. ROSES specifies each schedule as a chronological list of "assignment-starting" triplets *t,p,e* where *t* is the time for processor *p* to start task *e*. For convenience, the list also includes triplets indicating the starting times of idling periods, using $e = 0$ to indicate idling. With this inclusion, the schedule need not be accompanied by the information *L(e)*; because it is understood that unless the goal has been reached when a processor finishes an assigned task, there will always be a new assignment listed for it, either with $e > 0$ or $e = 0$.

A ROSES schedule always satisfies all the precedence constraints and all the processor-availability constraints. However, the schedule may or may not be complete (where "complete" implies reaching the goal).

In specifying a schedule, ROSES tags each successive triplet with an ordinal number spt (for "selection point"). When constructing a schedule, ROSES works forward in spt and time. That is, ROSES make its selections for *[t,p,e]* in order of increasing spt and nondecreasing *t*. Using curly brackets to indicate a set, we write the schedule as

$$\{[t,p,e]\} \equiv \{[t_{(spt)}, p_{(spt)}, e_{(spt)}] for spt = 1,2,3,\ldots\} \quad .$$

Within Eq. (1) we have used — for a particular assigned processor and its associated task (or idling) the same symbols p and e which were introduced earlier as generic running indices. In later discussions too we shall use p and e sometimes as special, sometimes as generic. Usually the context will make our intended meaning clear (but where clarity does require different symbols, we shall use different symbols).

Two features of Eq. (1) lead to possibilities for the construction of some essentially redundant schedules. These features are: the use of processor indices p, and the use of selection-point indices spt even when a subseries of triplets has the same t. (The simplest p-associated example is that two ROSES schedules would be essentially redundant if they differed from each other only by an overall permutation of their processor indices.)

However, ROSES successfully avoids constructing such redundant schedules. To do this, ROSES uses special redundancy-avoiding procedures to sequence tasks and to sequence processors. These redundancy-avoiding procedures pervade the ROSES method; they are intertwined with the procedures used to satisfy basic constraints and to apply heuristics; and they involve the same graph-theoretic techniques and data structures that ROSES uses to satisfy the constraints and to apply heuristics.

In this report we want to most strongly emphasize, as basics, the procedures for satisfying constraints and applying heuristics. Accordingly, we want to avoid cluttering our explanations of these basics with numerous details about the intertwined redundancy-avoidance procedures. Still, we want to make clear that – and how – redundancies are avoided. As a compromise, we have put some of the redundancy explanations into Appendices A and B.

III.3. STRUCTURE OF A ROSES SEARCH

Construction of a single schedule proceeds according to the following heuristic rules and conventions: At $t = 0$ (the start of the schedule), each of the N_p processors is given an assignment — either one of the tasks $e > 0$, or an idling assignment $e = 0$. For this initial set of triplets, the processors are assigned in order of p. At $t = 0$, the only tasks that can be assigned are those which have no precedence requirements. Partly for heuristic reasons, and partly to avoid redundancies, tasks are always assigned in a preference-order that is strongly influenced by critical-path considerations. After $t = 0$, assignments are made at, and only at, task-finishing times. These are the only times when processors become free (because they have just finished tasks), and when further tasks may become assignable (because a just-finishing task may complete their precedence requirements). Whenever a processor finishes a task, that processor is immediately given a new e-assignment, either $e > 0$ or $e = 0$. Furthermore, at that same task-finishing time t, each processor that has been idling is given a newly starting assignment, either $e > 0$ or $e = 0$. If, at a given task-finishing time t there are several different processors which finish their tasks or have been idling, then ROSES gives them assignments with simultaneous starting times but successive spt indices. In a complete schedule, the last time t specified is "TIME-to-GOAL," which coincides with the finishing time of the last-to-finish task.

In the course of searching for an optimal solution, ROSES constructs at least one complete schedule — and sometimes many alternative schedules. Table 1 shows such a set of alternative schedules. In this table, each capital letter represents an

entire assignment-triplet *[t,p,e]*. We comment now on two qualitative features which Table 1 prominently displays:

a. Some schedules are abandoned before being completed all the way to goal.

b. Two successively constructed schedules generally share their history, up through some spt.

Table 1. Schematic Example of a Set of Schedules Constructed Successively by ROSES

spt	Schedule* #1	Schedule* #2	Schedule* #3	Schedule* #4
1	A	A	A	A
2	B	B	B	B
3	C	C′	C′	C′
4	D	D′	D″	D″
5	E	E′	E″	E″
6	F	(abandoned	F′	F″
7	G	after	G′	G″
	(continues	spt=5)	·	(abandoned
	to goal)		·	after
			(continues	spt=7)
			to goal)	

*Below, each capital letter represents an assignment-starting triplet $[t, p, e]$.

Both (a) and (b) are easily understood in terms of ROSES using a modified depth-first search method enhanced with some A*–like features (e.g., time bounds) and some additional criteria for backtracking. Along each path in the search tree, successive vertices, and the branches stemming from them, are labeled with the ordinal numbers spt. At each vertex the branches correspond to allowed alternatives for a triplet *[t,p,e]* to be added to the series of triplets forming the path leading to that vertex. (We use the word "branch" to mean the path-segment between two successive vertices.) This search-tree picture will be used in comments on (a) and (b).

Concerning (a), i.e., the fact that some schedules are abandoned before being completed all the way to goal: ROSES aborts a schedule upon finding evidence that if the schedule were continued all the way to the goal, that completed schedule would not be better than — i.e., would not have a shorter TIME-to-GOAL than — the best complete schedule previously found. Evidence on this matter is continually sought: for (in the A* spirit) ROSES calculates, at each spt, several different kinds of time bounds on the TIME-to-GOAL for a schedule incorporating the history-through-spt of the schedule currently under construction. In addition, ROSES uses

some other, heuristic criteria for deciding whether or not to continue the schedule under construction.

Concerning (b), i.e., the fact that two successively constructed schedules generally share their history, up through some spt: ROSES generally constructs a new schedule by altering only some later portion of the preceding schedule. The similarity of consecutive schedules in Table 1 comes about from selecting and traversing a particular branch $[t,p,e]$ at an spt vertex in the search tree, then deciding not to continue the current schedule past that branch, and then backtracking in the search tree to the same spt vertex or to a yet earlier vertex in preparation for proceeding forward through a previously untraversed branch.

ROSES tends to backtrack as little as possible — or, to put it positively, ROSES tends to press forward toward the goal as much as possible. This "pressing-forward" approach is in line with the choice of depth-first style for ROSES. The depth-first choice means that, unless there is evidence which warrants abandoning a partially complete schedule, ROSES works to complete that schedule before switching to construction-work on a different schedule. Furthermore, after a schedule is completed or abandoned, ROSES constructs the next trial schedule not by starting anew at $t = 0$, but instead by backtracking in the search tree only as far as the latest previous vertex where (according to saved information) at least one as-yet-untried branch remains and is worth traversing. Then the new schedule starts with a portion identical to the early portion of the preceding schedule, but continues differently in moving toward the goal from the vertex to which the search backtracked.

This style of searching, i.e., pressing forward, tends to maximize the speed of accumulating pertinent information about completed (and possibly near-optimum) schedules. The accumulated information is put to good use by ROSES, in evaluating later-constructed partial or full schedules.

After completing a schedule (i.e., reaching the goal), ROSES stops searching if the TIME-to-GOAL is judged sufficiently satisfactory, or if the search tree has been explored as fully as the ROSES heuristic rules dictate. If neither of these conditions holds, then as noted above, ROSES backtracks to the last previous search-tree vertex where any as-yet-untried branches exist and are worth traversing.

The search tree is constructed so that ROSES can form only schedules which satisfy the precedence constraints and processor-number constraints. Furthermore, the search tree is constructed so that it disallows several kinds of essentially redundant schedules.

However, the details of the search tree are not known in advance of traversing the tree. Only after ROSES reaches a search-tree vertex for the first time, by forward-traversal, does ROSES determine the further allowed triplet-choices and so construct the search-tree section just ahead of that newly reached vertex. At each vertex spt, the set of allowed triplet choices depends strongly on the series of triplets chosen at past selection points of that particular schedule's history. During forward search, ROSES continually accumulates and updates information depending on the schedule-under-construction, in order to determine the characteristics of the next search-tree vertex with all its emanating branches. Besides identifying these branches, ROSES determines a preference-order for them. These determinations are made before traversing any one branch at that vertex. In short: for a given spt,

ROSES works to "nominate" a list of candidate triplets, and gives them a preference order, before evaluating each particular candidate in turn.

When doing the work of nominating search-tree branches, ROSES is much concerned with questions of readiness. Which tasks are ready to be assigned, because their precedence requirements have been satisfied? Which processors are ready to accept new assignments because they have finished their previous assignments? When doing the work of evaluating search-tree branches, ROSES is much concerned with time bounds. The needed information on readiness and time bounds is stored in terms of data structures.

Our next four sections deal in turn with the search tree, readiness, time bounds, and data structures.

III.4. PROPERTIES OF A ROSES SEARCH TREE

Many of our later explanations are cast in terms of search-tree construction and traversal. To provide background for those explanations, this section reviews and extends our foregoing remarks about ROSES search-tree features and terminology.

As noted earlier, we use the word "branch" to mean the search-tree path-segment connecting two vertices. ROSES constructs each search-tree part — vertex plus stemming-out branches — only if and after ROSES has (1) traversed the path leading to that vertex, and (2) decided that the traversed path is sufficiently promising to extend and traverse further.

- Basics: Vertices, Branches, Schedules, and Selection-Point Indices

Figure 1 shows a very limited portion of a ROSES search tree.

Each search tree vertex is labeled with an index spt (as well as other descriptors). Similarly, each branch is labeled with an index spt. As spt increases, time t is nondecreasing. Our convention is that spt increases in the upward direction. Thus, a "forward-going" search-tree path proceeds upward. A forward-going search-tree path, if it starts from the bottom of the tree, corresponds to a schedule. The successive branches within that path correspond to the assignment-starting triplets $[t, p, e]$ of the schedule. The bottom point of the search tree corresponds to $t = 0$ with all processors idling. It is a vertex tagged with the selection-point index $spt = 1$. Every schedule-path starts there, and the further vertices along each schedule-path are tagged with successively increasing selection-point indices $spt = 2, 3, 4, \ldots$, etc.

A schedule-path is complete if its top vertex implies attainment of the goal. In the search tree, many different vertices, each at the top end of a separate path, can individually imply attainment of the same goal (all tasks completed). Each separate complete path implies a different way of reaching the goal.

Although many assignments may be made at the same time t, each branch corresponds to only a single assignment-starting triplet $[t, p, e]$. For example, at $t = 0$ all of the N_p processors are assigned; but each of the branches stemming from the $spt = 1$ vertex represents the initial assignment for one particular processor, not the entire set of initial assignments to all processors.

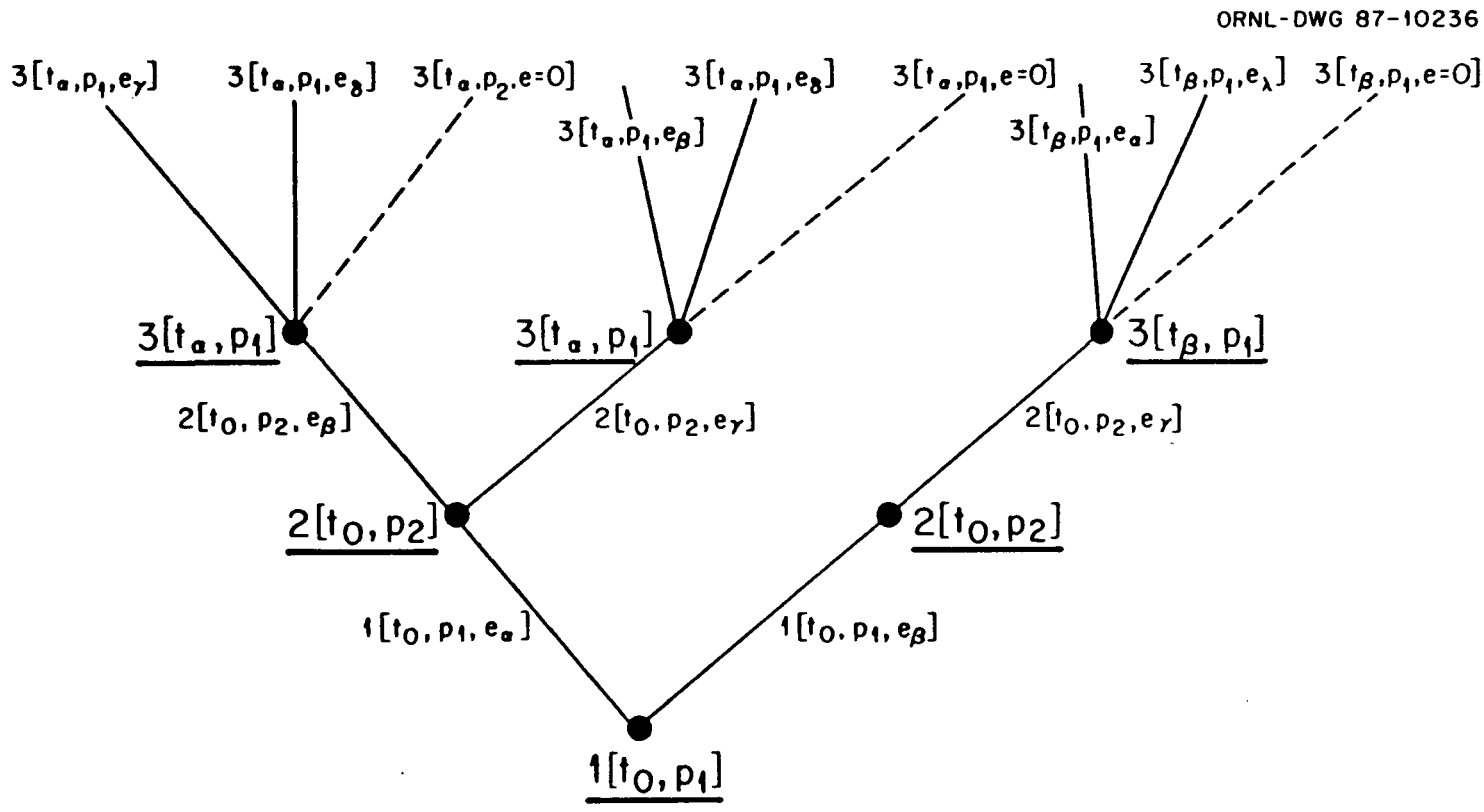

Fig. 1. Portion of a ROSES search tree, for a two-processor case. Vertices are marked with large underlined labels $spt[t,p]$. Branches are marked with small labels $spt[t,p,e]$. Each illustrated schedule-path shows an initial assignment for p_1, then an initial assignment for p_2, and then a reassignment for p_1 when p_1 completes its initial assignment.

If a branch stems from a vertex tagged spt_c, then that branch too is tagged spt_c. This convention — relating vertex-indices to branch-indices — is convenient for correlating our description with the indexing scheme used in the main control structure of the ROSES code. A ROSES search progresses by repeatedly passing through that main control structure. Each pass is associated with "current" entities which we refer to here by using subscripts c.

Thus, the beginning-of-pass, forward-going path starts from $spt = 1$ and goes up to a vertex tagged spt_c. That is, the beginning-of-pass schedule-path corresponds to a schedule incorporating (spt_c - 1) triplets. The branches stemming from the current vertex spt_c correspond to candidates for a triplet to be added to the current (spt_c - 1)-member schedule, so as to form an spt_c-member schedule.

- Branches Stemming from the Same Vertex

Because of the heuristic rules and conventions described early in Section III.3 (and also, because of conventions and procedures used to avoid redundancies): All nominated triplets (branches) stemming from the same search-tree vertex have the same $[t, p]$ combination; it is only in e–value that they differ. Accordingly, a vertex may be labeled with $[t, p]$ as well as with spt. Furthermore, when discussing a fixed search-tree vertex, we sometimes think in terms of alternative e–choices, rather than alternative $[t, p, e]$ choices.

At each vertex there is one stemming-out branch which represents idling, $e = 0$. For all other branches, the e represents a task, for which $e > 0$. Every one of these task branches refers to a task that is "ready" — where a ready task means a not-yet-started task whose precedence requirements are satisfied. The task-branches at each vertex are arranged, left to right, in order of decreasing f, L favor. (The rules used to evaluate "f, L favor" will be described later in this report.) The idling branch is placed at the extreme right, for it is the "least favored" of all the branches stemming from that vertex. For brevity, we use terms such as "most favored" to refer to e-choices in this order: the ready tasks in decreasing order of f, L favor, and then the idling choice $e = 0$.

What is the import of "favor"? When ROSES is forward-searching from a vertex spt, ROSES always traverses the most favored (leftmost) branch that has not yet been traversed. Then on successive revisits to a vertex (revisits allowed via backtracking), ROSES traverses successively less favored branches.

The task-branches stemming from a vertex $[t, p]$ do not always represent the entire set of tasks ready at t. Some of those ready tasks may be excluded because of redundancy-avoiding task-ordering conventions. (Discussions of this matter appear in several places within this report, e.g., in Appendices A and B). However, whenever the t of a vertex spt is different from that of its preceding vertex spt - 1), then the branches stemming from the vertex spt do correspond to all the tasks ready at t, plus the idling branch.

- Specification of Vertices and Branches

The combination spt,$[t, p]$ is not generally enough to uniquely specify a vertex. There could be several vertices, in different parts of the search tree, that are identical in their combination $spt, [t, p]$. Similarly, the combination $spt, [t, p, e]$ is not generally enough to uniquely specify a branch.

Even within a given path, there may be multiple vertices with a given $[t,p]$; this can occur because zero-length tasks are allowed.

However, when discussing a fixed schedule-path we can properly identify a unique vertex by referring to "the *spt* vertex," and we can properly identify a unique branch by referring to the "*spt* branch" or the "*e*-branch" (where here *spt* and *e* imply special values, not generic variables). These references are precise because, within a given schedule-path, there is no more than one vertex tagged with a particular value *spt*, no more than one branch tagged with a particular value spt, and no more than one branch for a particular value *e*.

- The "Assignment-Finishing" Set

One can think of each branch as representing a discrete triplet-assignment event in the schedule; or alternatively, one can think of the *vertices* as the discrete events, and the branches as representing a span of changing time and other coordinates. In either case: Aside from the branch's assignment-starting triplet $[t,p,e]$ and *spt* there are additional attributes implied by a branch and its position.

One such additional implication is the assignment-finishing set $[t_\phi, p_\phi, e_\phi]$ characterizing the vertex which constitutes the upper end-point of a branch $spt, [t,p,e]$. The finishing time t_ϕ and the finishing processor p_ϕ are to be identified with the pair $[t_{(spt+1)}, p_{spt+1)}]$ at the vertex following spt. The finishing time t_ϕ is the earliest time, at or after t, that any previously assigned task finishes. The finishing processor p_ϕ is a particular processor finishing its assignment at t_ϕ — and the branch's starting processor p does not usually fit this description. If there is more than one processor finishing at t_ϕ, then the particular p_ϕ associated with a branch is determined by a set of redundancy-avoiding conventions described later in this report. Once p_ϕ is determined, e_ϕ is determined; for e_ϕ is the task or idling assignment finishing at t_ϕ on p_ϕ.

Thus, a given branch (and/or its pair of bounding vertices) does not generally represent the start and finish of one task. The finishing processor p_ϕ is not generally the same as the starting processor p; the finishing task e_ϕ is not generally the same as the starting task e; and the finishing time t_ϕ is not generally the same as the finishing time $t + L(e)$ of the starting task. (However, it could happen, in some cases, that $e_\phi = e$ and $p_\phi = p$ and $t_\phi = t + L(e)$.

- Terminology: Planning Schedules and Executing Schedules

Here we describe a language convention that we have already been using: Whether or not we talk in search-tree terms, we often use jargon that ignores the distinction between scheduling (i.e., planning) a set of tasks, and actually executing those tasks. For example, if during schedule-construction a task e has just been assigned to start at time t we may say that task e "has started." Similarly, if during schedule-construction ROSES is at the point of considering what other tasks can be scheduled after completion of task e, then we may say that task e "has been completed."

- More Terminology: Traversing a Branch; Adding a Triplet; etc.

Terminology is important because we want to have different descriptive phrases, with differing defined meanings, to correlate with different kinds of operations within the ROSES algorithm.

The phrase "traversing a branch" is interpreted as actually adding that branch's triplet $[t, p, e]$ to the schedule. Elaborations follow.

There is a clear distinction between constructing the search tree and traversing the search tree. Constructing the search tree entails determining the set of branches $[t, p, e]$ stemming from a vertex. Traversing the search tree entails trying out a particular branch as an addition to the current schedule-path: i.e., adding a triplet to the schedule, and investigating whether the resulting extended schedule-path is promising. Thus, traversing the search tree is equated with constructing a schedule; and we treat the following notions as essentially synonymous:

- traversing a branch;
- adding a triplet;
- making an assignment;
- extending a schedule;
- evaluating a branch, triplet, assignment, or augmented schedule.

There are many distinctions left to make, and worth making. One reason for defining terminology carefully is that we want to have different descriptive phrases, with differing defined meanings, to correlate with different kinds of (sub)procedures in the ROSES code. For example, when we describe features of the ROSES coded algorithms, we shall be concerned with the distinctions among:

1. Constructing all the branches at a search-tree vertex;
2. Considering whether a vertex may have, stemming from it, any as-yet-untraversed branches that are worth traversing;
3. Traversing a branch and then deciding that the resulting schedule-path is worth continuing further;
4. Traversing a branch and then deciding that the resulting schedule-path is not worth continuing further.

In (2) and (4) the judgments "not worth traversing" and "not worth continuing" may involve TIME-to-GOAL considerations, or redundancy considerations.

- Limitation to Constraint-Satisfying, Non-Redundant, "Promising" Schedules

The ROSES method for constructing its search tree leads to these properties:

- The search tree's branches are limited so that, though taking any search-tree path, ROSES can construct only those schedules which satisfy the input problem's processor-number constraints and precedence constraints.
- Furthermore, the search tree's branches are limited so that, if ROSES traverses each branch no more than once, ROSES avoids the construction of several general kinds of essentially redundant schedules. (See Appendix A.)

In other words: In order to limit its attention to schedules which are constraint-satisfying, and which are free from some types of redundancy, ROSES directly limits its construction of search-tree branches.

In contrast, the way that ROSES limits its attention to schedules with promising time characteristics is by limiting its traversal of already constructed branches. In addition, some kinds of redundancy are avoided by limiting traversal. (See Appendix B.)

III.5. TASK GRAPHS AND READINESS

Precedence constraints, and the readiness of tasks, are conveniently considered in terms of directed acyclic graphs (dags).[34] A dag has directed edges connecting its vertices. Any problem of the kind treated by ROSES (see Section III.2) can be expressed in terms of a dag. Each task is represented by one edge e in the dag, and each precedence constraint by one vertex v of the dag. The tasks' time-lengths are brought into the representation by associating a non-negative number $L(e)$ with each edge.

To immediately assure the required isomorphism, the ROSES-problem precedence constraints must be stated so that each predecessor-task is disjoint from all others, and each successor-task subset is disjoint from all others. However, if such disjointness does not hold in the problem as initially stated, then it is always possible, by adding some zero-length tasks, to pose an equivalent problem with the desired disjointness. Hereafter we shall assume that the task-set has indeed been defined so that the desired disjointness exists; then an isomorphic dag exists.

The dag for a ROSES scheduling problem is called a task graph. Figure 2 diagrams a simple task graph for ROSES.

The goal is drawn at the task graph's top. As already noted, the graph's edges represent tasks. Computational tasks are drawn as solid-line edges, message-passing tasks as dashed edges. The diagrammed edge-lengths are not generally drawn to scale with $L(e)$. The graph's vertices imply ROSES-type precedence constraints. For example, in Fig. 2 every one of the tasks e_{12}, e_{13}, e_{14} must be completed before task e_7 can start; and task e_7 must be completed before any one of the tasks e_{17}, e_{18} can start. In short, the vertices represent AND gates.

Every task-graph vertex is uniquely indexed, and each edge extends between two definite vertices. Thus, the vertex-indices can serve as labels for the edge's beginning-point (tail) and finishing-point (head). Edges e_{12}, e_{13}, e_{14} are called fan-in edges to vertex e_{14}. Edges e_{17} and e_{18} are fan-out edges from e_{15}.

A vertex is called "ready" if and only if all its fan-in edges have been completed. The goal is reached when it, as a vertex, is ready.

A task is called ready if and only if all its precedence requirements have been satisfied and the task has not yet been started. Therefore an edge is called ready if and only if (1) it is a fan-out edge from a ready vertex and (2) its corresponding task has not yet been started.

A complete set of the graph-attributes needed by ROSES consists of an input-data list specifying, for each task e, its length $L(e)$ and its head-tail pair $[u, v]$. At the

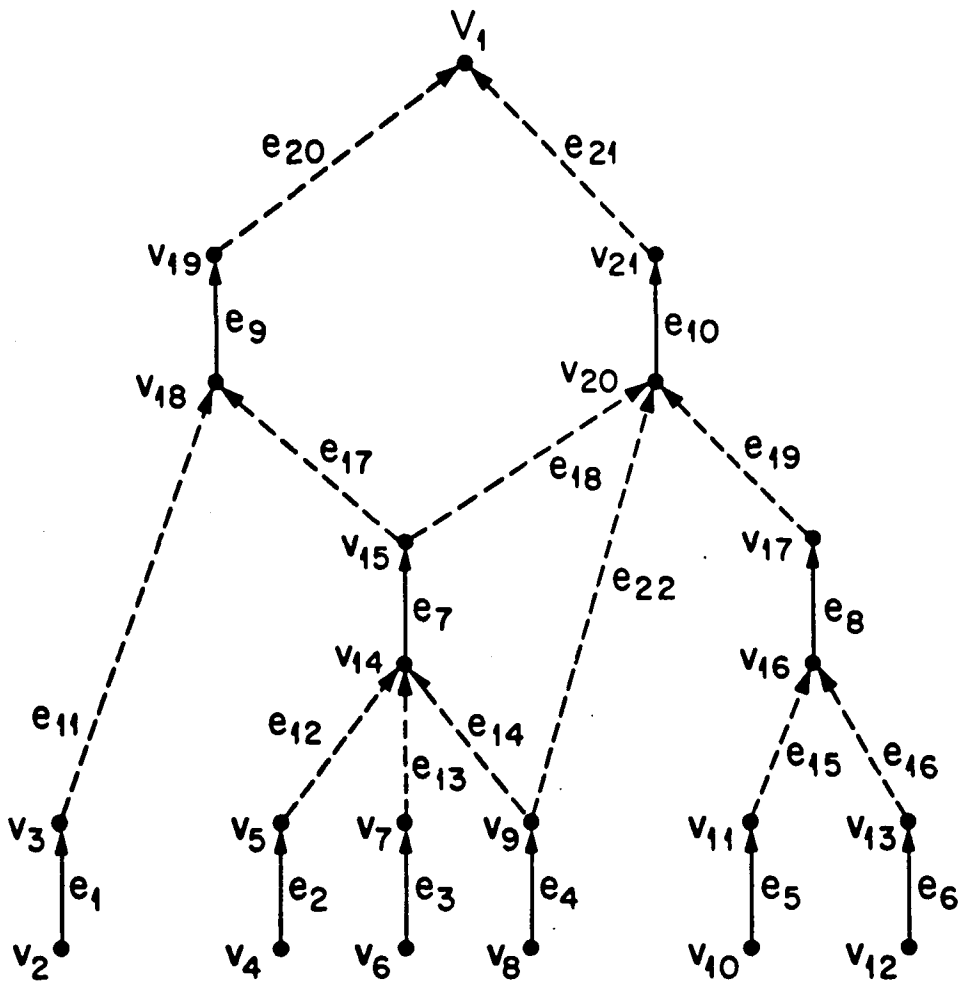

Fig. 2. A ROSES Task Graph. Graph edges denote tasks; solid lines represent computational tasks, while dashed lines indicate communication tasks. Graph vertices define synchronization points for the precedence constraints. Vertex V_1 symbolizes the goal node.

start of an input problem, when no tasks have yet been performed, there must be some tails which are ready. These are tails without fan-in edges. Accordingly, all their fan-out edges are ready.

Each message-passing task has only one predecessor task (always computational) and one successor task (always computational). In the special case that all the precedence requirements between computational tasks are expressed in terms of message-passing tasks, there is automatic fulfillment of the requirement that all predecessor-task subsets be disjoint from each other and all successor-task subsets be disjoint from each other.

The task graph shown in Fig. 1 is of the special kind just mentioned: i.e., all precedence requirements are expressed in terms of message-passing tasks. Consequently, Fig. 1 has the following properties: No computational task shares its tail vertex with any other task having the same head; and along any task-graph path, computational tasks alternate with message-passing tasks. (However, ROSES is not restricted to such cases.)

The task graph should not be confused with the search tree. Some examples of important differences follow:

- A complete schedule would involve every edge in the task graph, but only one path in the search tree.
- The task graph does not deal directly with processors; the search tree does.
- The task graph does not incorporate line-segments representing idling; the search tree does.
- The task graph is not a tree. A task-graph vertex can have many fan-in edges, but each search-tree vertex has only one lead-in branch.
- In the task graph, a line-segment with its two bounding vertices always represents a single task e starting at the tail vertex and finishing at the head vertex. In the search tree, a branch with its two bounding vertices implies, among other things: one task (or idling assignment) starting at the smaller-spt vertex, and a generally different task (or idling assignment) ending at the larger-spt vertex. The search-tree task that starts at the smaller-spt vertex may not finish until many vertices beyond its starting vertex.
- The task graph provides a convenient way to exhibit the state of the multitask job. For example, each task-edge could be colored red part-way up, to indicate its stage of completion. The search tree does not offer that particular visual convenience; instead it offers an analogous way to exhibit the state of the search.
- The task graph is constructed wholly, at the very beginning of an overall ROSES search. The search tree is constructed (branch-cluster by branch-cluster) as the overall search progresses.

ROSES uses the task graph to help construct the search tree. By combining the dynamic information about a current schedule, with the static information of the task graph, ROSES determines which vertices are ready and which edges are ready. Then ROSES uses this information in identifying the set of *e*-choices at the next

search-tree vertex. The task graph is also used in computing critical-path data that affects the sequence in which paths of the search tree are traversed (i.e., the sequence in which alternative schedules are constructed).

One kind of "readiness" information not related to the task graph (but needed in search-tree traversal) is the readiness of processors. A processor is "ready" if it is free to accept a task-assignment ($e > 0$) — i.e., if it is not still occupied with an earlier-assigned task. All processors are ready at $t = 0$.

Hereafter, we use the words "task" and "edge" interchangeably. Note that an edge (task) always has $e > 0$. Only idling assignments have $e = 0$.

III.6. TIME BOUNDS AND HEURISTICS

In ROSES, time-bound considerations play an essential role. They are used in two broad ways: to sequence the code's consideration of e-choices at spt (so that the most promising choices are considered first), and to evaluate partially completed schedules (so as to decide whether a partially constructed schedule is promising enough to be worth continuing).

The basic time-bound ideas may be stated in terms of two simple lower limits on the total TIME-TO-GOAL for a given problem. One of the two lower limits, T_o, is calculated by ignoring precedence constraints and ignoring some incommensurable-length effects, while taking into account the real (limited) number of processors. The other lower limit, T_∞, is calculated by assuming that the precedence constraints do apply, but that the number of processors is infinite. The two lower-bound values are (as discussed below):

$$T_o = \text{(sum of all task lengths)/(number of processors);} \tag{2}$$

$$T_\infty = \text{critical path length to goal, in the task graph: i.e., maximum end-to-end length of any complete precedence-chain-of-edges in the task graph.} \tag{3}$$

In (3), a "complete precedence-chain-of-edges" is a series of edges which in the task graph would make a continuous path starting from the tail of an edge without precedence requirements and going all the way to the goal.

In (2), the lower bound T_o is just the TIME-to-GOAL that would apply if no processors were ever idle at any time from the schedule's start ($t = 0$) until its end (t = TIME-to-GOAL). However, idle time may be unavoidable – unavoidable because of the precedence constraints, and/or because there is not way to partition the total set of tasks e into N_p subsets each having the same subset-sum of time-lengths.

Clearly, the problem may be such that neither T_o nor T_∞ is attainable. T_o may exceed T_∞, or vice versa; that depends on the input problem. The larger of the two bounds is the more useful. If a completed schedule's TIME-to-GOAL is equal (or not much greater than the maximum of T_o and T_∞, then that schedule is optimal (or

near-optimal). However, if a completed schedule's TIME-to-GOAL is considerably larger that the maximum of T_o and T_∞, then that solution may or ma not be considerably larger than the true optimum. Further information is needed, to determine how much the true optimum value deviates from the maximum of T_o and T_∞. In cases where neither T_o nor T_∞ is attainable, a better lower bound may be determined by adding and utilizing more information about the input problem.[13]

Relatedly, lower bounds may be determined for the possible completed forms of a partially constructed schedule, by using information about the so-far-constructed portion of that schedule. ROSES continually makes such determinations. That is, during construction of a schedule, ROSES continually uses variations of Eqs. (2) and (3) to determine lower bounds on TIME-to-GOAL for the feasible forms of the partially completed schedule. For example, a very simple variation of (2) involves taking into account the idling time that has already been assigned, within the currently incomplete schedule under consideration.

Such lower bounds are determined, during construction of a schedule, at each spt prior to goal-reaching. By comparing lower bounds for the completed form of the current schedule, with the TIME-to-GOAL of the best earlier-found solution, ROSES decides whether the partially constructed schedule is worth continuing or should be abandoned. As indicated just below, there are two different sets of time-bound tests for partially completed schedules. In search-tree terms, there are two qualitatively different sets of search-tree points where straightforward time-bound tests may lead to decisions-to-abandon.

1. In some cases, time-bound tests show that the t at a search-tree vertex $[t, p]$,spt is unsatisfactory when that t is combined with other implications at the vertex. In such cases ROSES backtracks without bothering to traverse any as-yet-untraversed branches stemming from the vertex. The vertex is said to be "unsatisfactory in t." For a vertex, satisfactoriness in t depends not only on t itself and the schedule up to vertex spt, but also on the best TIME-to-GOAL previously found. Consequently, a vertex which is "satisfactory in t" at one visit may be "unsatisfactory in t" at a later visit; that can happen if a better TIME-to-GOAL was found in between the two visits.

2. In other cases, the vertex itself is judged satisfactory, but time-bound testing indicates lack-of-promise after a particular branch has been traversed — i.e., after a particular triplet has been added to the schedule. In such cases, one can say that the schedule was considered sufficiently promising to continue past the (spt − 1) branch, but is found too unpromising to continue past the recently added spt branch. In such cases, ROSES associates the lack-of-promise of the schedule with the addition of the triplet at spt, and so ROSES next investigates the remedy of replacing the offending *spt*-triplet $[t, p, e]$ with an alternative branch stemming from the same vertex $[t, p], spt$. If no as-yet-untraversed branches remain at that search-tree vertex, then ROSES back-tracks further.

If these backtracking rules cause ROSES to try backtracking past spt = 1, then the tree has been traversed as fully as the ROSES heuristics dictate. In that case, ROSES stops searching.

Suppose the goal is reached. Then if TIME-to-GOAL is judged sufficiently close to a known overall lower bound (say, T_o), ROSES decides to stop searching. Otherwise (unless other reasons for stopping exist), ROSES backtracks to continue its search.

The above descriptions indicate how ROSES uses the time-bound notions to decide whether to backtrack. Furthermore, ROSES uses a heuristic related to T_∞ to give an *apriori* preferred sequence to the evaluation of edge-choices at each spt. Specifically: At each vertex of the search tree, ROSES considers candidate edge-branches in decreasing order of

$$\begin{aligned} \text{f(e)} = {} & \text{critical task-graph-path-length to GOAL beginning with edge } e\text{:} \\ & \text{i.e., the maximum end-to-end length of any precedence-chain-} \\ & \text{of-edges in the task graph, starting from the tail of e and} \\ & \text{incorporating } e \text{ itself and going all the way to the goal.} \end{aligned} \tag{4}$$

As noted, the larger that $f(e)$ is, the more favored the candidate edge e. Furthermore, if two candidate edges have the same $f(e)$ but different time-lengths $L(e)$, then the shorter edge is considered more favored. Candidates which are equal in both f and L are considered in an arbitrary order (sometimes depending on other features of the calculation being performed). The entire scheme (including its arbitrary-order part) is referred to as "the f, L heuristic"; it arranges tasks e in order of decreasing "f, L favor."

As to the non-task choice, e=0: At each spt vertex ROSES places at the end of the candidate-list — lower in favor than all of the available edges representing tasks — the idling candidate e=0. This lowest-in-favor position is in line with the lower bound T_o of (2), because there will be a tendency for the total idling time to be minimal of idling is avoided when individual assignments are chosen.

The "order of considering branches" concerns backtracking. If a search-tree vertex is found to be "unsatisfactory in t," then there are no consequences from the favor-sequence of its stemming-out branches. But for the moment, let us suppose that the search-tree vertex is not unsatisfactory in t (and not unsatisfactory in any other way that makes ROSES avoid traversing the as-yet-untraversed branches at that vertex). Then it can be said that: At a given search-tree vertex spt, ROSES considers the most favored e-choice first — i.e., when first visiting that vertex. Then on successive return visits to that vertex (reached through backtracking), ROSES considers successively less favored e-choices.

In short, at a given satisfactory search-tree vertex, ROSES always traverses the most favored as-yet-untraversed branch. Within this procedure the f, L heuristic has been crucial to the success of ROSES.

ROSES also uses an extension of the above-described f, L heuristic in order to classify some kinds of schedules as "not worth constructing" even before testing those schedules in terms of time bounds. This extension involves a function called the delay function $D(e)$ which characterizes a given schedule and is defined for all edges e. Specifically, we define $D(e)$ to be the number of search-tree vertices, within the schedule-path, at which e characterizes the most-favored (leftmost) branch but that branch is not incorporated in the schedule-path. In other words: $D(e)$ is the number of vertices, within the schedule-path, at which task e was the most favored stemming-out edge-choice but was "delayed" rather than selected. ROSES immediately classifies as "unsatisfactory in D," and therefore "not worth constructing," any schedule which would have any delay function $D(e)$ greater than D_{MAX}, where D_{MAX} is an input parameter called "the maximum delay."

It is easy for ROSES to avoid constructing such schedule-paths. Any schedule with any $D(e') > D_{MAX}$ must incorporate an spt'-member schedule-path such that: $D(e') = (D_{MAX} + 1)$, and e' characterizes the most favored (leftmost) edge at that vertex, and all the branches except the leftmost one would make schedules with $D(e') = (D_{MAX} + 1)$. Therefore, to avoid all schedules unsatisfactory in D, ROSES needs only to do this: Avoid adding any branch except the leftmost one, at any vertex where adding the leftmost branch makes a schedule with $D(e') = D_{MAX}$. After adding that leftmost branch, ROSES marks the vertex as "unsatisfactory in D," or more simply as "having no satisfactory as-yet-untraversed branches"; then if and when that vertex is revisited (through backtracking) ROSES avoids adding branches at that vertex and instead backtracks further.

Note that this $D(e)$ procedure for limiting search does not simply put a constant numerical restriction on the branching degree at each vertex (as in "beam" searching). Instead, the $D(e)$ procedure uses detailed characteristics to selectively reject search-tree paths. At present the it D(e) procedure is the only one, in the ROSES repertory, which excludes schedules that are not surely known to offer zero possibility of excelling the best complete solution found earlier in the search.

This report-section, Time Bounds and Heuristics, has presented the general principles of ROSES time-bound testing and described some related heuristics. Some of the *specifics* of the time-bound tests will be described in Section IV.5 in connection with outlines of coded algorithms. Among the further time-bound-related heuristic features of ROSES (beyond those described above), there are:

- a modified dynamic-programming scheme, and
- a way to characterize "precedence-bottleneck" features of the task graph, and to take them into account so as to calculate a closer lower bound than T_∞.

III.7. ROSES DATA STRUCTURES

In the following we present examples of how ROSES expresses and stores, in conveniently accessible forms, key information needed to conduct a successful search along the lines described in earlier sections. Much use is made of abstract data types,[34] involving such complex data structures as

- doubly linked lists describing sequenced sets of data blocks, (with sequence itself carrying important information);
- array implementations of embedded stacks;
- circular-array implementation of a queue.

In describing the ROSES framework, what we want to convey are the general notions of: what information each data structure stores, and what sorts of operations can access that information. We shall describe ROSES structures in terms of simple specific Fortran arrays and variables; then readers themselves can abstract the general notions. The actual ROSES-code arrays and variables differ in detail from the Fortran entities specified here. However, the general notions are the same. Appendix D gives further information about the correspondence between this report's entities and ROSES code entities.

III.7.1. The Initial Task-Graph Structures ITG and G

The static data structure ITG holds all the information necessary to specify the initial task graph. The principal component of ITG is itself a structure, G. Specifically: The structure G comprises a series of 4-word substructures called quartets, each pertaining to one of the N_e edges e. The entire series is arranged in order of increasing e (but that order need not bear any relation to the topology of the task graph). Within each quartet the four words store, respectively:

1. the tail vertex of e
2. a pointer to the last-previous quartet associated with the same head as e has
3. the head vertex of e
4. a pointer to the last-previous quartet associated with the same tail as e has.

Thus, G stores information that would be embodied in four coupled arrays:

1. tail (e)
2. previous-e-for-same-head(e)
3. head(e)
4. previous-e-for-same-tail(e)

The array names are largely self-explanatory, but we give one precise definition as an example: The element "previous-e-for-same-head(e)" is a pointer to the largest index e' such that head(e') = head(e) and $e' < e$. If there exists no such index e', then the element "previous-e-for- same-head(e)" holds a special value signaling that it is nonexistent. Similar comments hold for "previous-e-for-same-tail."

Using these arrays, we write the structure G as

$$\begin{aligned} \mathrm{G} = \{&[\text{tail}(e), \text{previous-}e\text{-for-same-head}(e), \\ &\text{head}(e), \text{previous-}e\text{-for-same-tail}(e)] \text{ for } e = 1, 2, 3, ...\} \end{aligned} \tag{5}$$

As the foregoing description implies, the structure G is a series of quartets but incorporates many linked lists. For each task-graph vertex v, there are two lists — one backward-linked list-of-quartets describing all the fan-in edges to v, and another backward-linked list-of-quartets describing all the fan-out edges from v. Each quartet is part of two different linked lists — the fan-in list associated with its head-vertex v_h, and the fan-out list associated with its tail-vertex v_t. Thus, G is a doubly-linked structure which does not really fit into the category of a doubly-linked "list." For example, instead of having one or two overall linked sequence(s), G has many linked strands each with its own linked sequence. This departure from simple linked-list linearity is of course a reflection of the fact that G represents a more complex set of edge relations — the dag[34] of the initial task graph.

We let the comprehensive initial task-graph structure ITG include not only G but also the time-length array $L(e)$ and the number-of-edges scalar N_e. Together, G and $L(e)$ and N_e completely specify the initial task graph. In fact, the pointer parts of G are not essential for defining the task graph; they are included because they facilitate locating edges with desired properties. For the same reason, ITG also includes:

$$\text{nfie(v)} = \text{number of fan-in edges to vertex } v \text{ of the task graph,} \qquad (6a)$$

$$\text{nfoe(v)} = \text{number of fan-out edges from vertex } v \text{ of the task graph,} \qquad (6b)$$

and

$$\text{hfie(v)} = \text{highest-indexed fan-in edge to } v, \qquad (7a)$$

$$\text{hfoe(v)} = \text{highest-indexed fan-out edge from } v, \qquad (7b)$$

By using G and the auxiliary parts (6), (7) of the structure ITG, ROSES can easily perform such subtasks as: Find all the edges fanning out from v.

Furthermore, the comprehensive initial task-graph structure ITG includes the critical-path-length array $f(e)$ of Eq. (4). Like the pointers, and like the arrays (6) and (7), the array $f(e)$ constitutes redundant data in the sense that it re-expresses information already within the more ordinary data set {head(e), tail(e), L(e)} that are input to describe the task graph. As noted earlier, $f(e)$ is part of a heuristic that is crucial to the success of ROSES. Therefore, even though $f(e)$ is a simple array rather than a many-feature data structure, it is a prime example of how ROSES converts "ordinary" information into especially useful, convenient forms.

III.7.2. The Processor-Readiness Structures PR and R

The processor readiness structure PR gives information about the spt-order in which processors are to be assigned. It also specifies each processor's current assignment e, and tells when each processor will finish its e and so be ready for reassignment.

The structure PR includes p, the scalar data element pointing to the "current processor" — i.e., the processor to be assigned, when ROSES is in the process of identifying the current triplet $[t, p, e]$ in preparation for adding it to the schedule. In search-tree terms: p is the processor index within the label $[t, p]$ of the current search-tree vertex.

Besides including the scalar data element p, the structure PR includes a complex structure R. The main points about R can be understood rather quickly if one understands and remembers some of the ideas discussed in III.3 — ideas regarding (i) triplet-assignment times t being coincident with task-finishing times, and (ii) the uniqueness of the sequence that ROSES uses when assigning different processors p,

in case these different processors are given triplet-assignments $[t, p, e]$ with the same starting time t.

In Eqs. (8) through (13) below, we define a set of four arrays equivalent in content to what R holds. Each of these arrays is indexed by an integer p' running from 0 to $(N_p + 1)$. For most of the N_p cases, i.e., for $1 \leq p' \leq N_p$, the index p' denotes a real processor and we have

$$\begin{aligned} \text{tready}(p') &= \text{ready time of processor } p', \\ &= \text{time at which } p' \text{ will finish its present assignment} \\ &= \text{time at which } p' \text{ should be considered for reassignment;} \end{aligned} \tag{8}$$

$$\text{evalue}(p') = e\text{-value presently assigned to processor } p'; \tag{9}$$

$$\begin{aligned} \text{pfwd}(p') = &\text{ pointer to that processor which would follow } p', \\ &\text{within a list of the } N_p \text{ processors arranged in} \\ &\text{order of nondecreasing tready}(p'); \end{aligned} \tag{10}$$

$$\begin{aligned} \text{pbak}(p') = &\text{ pointer to that processor which would precede } p', \\ &\text{within a list of the } N_p \text{ processors arranged in} \\ &\text{order of nondecreasing tready}(p'); \end{aligned} \tag{11}$$

However, when $p' = 0$ or $p' = (N_p + 1)$, the index p' does not denote a real processor. In each of these cases, only one of the four arrays has a defined element. We have

$$\begin{aligned} \text{pfwd}(p' = 0) = &\text{ pointer to that processor which would be first,} \\ &\text{within a list of the } N_p \text{ processors arranged in} \\ &\text{order of nondecreasing tready}(p'); \\ = &\text{ pointer to the processor that is first-in-line to be} \\ &\text{considered for reassignment;} \\ = &\ p, \text{ ``the current processor'';} \end{aligned} \tag{12}$$

$$\begin{aligned} \text{pbak}(N_p + 1) = &\text{ pointer to that processor which would be last,} \\ &\text{within a list of the } N_p \text{ processors arranged in} \\ &\text{order of nondecreasing tready}(p') \ . \end{aligned} \tag{13}$$

Definitions (10–13) mention a list of processors arranged in order of nondecreasing tready (p'). This list needs further description to specify the ordering of processors within any incorporated sublists composed of processors having identical tready(p'). To describe that ordering we describe how ROSES forms and updates the list. At $t = 0$ when the processors are all idling (before any task-assignments have been made), the processors are arranged simply in order of p'. Then shortly after each calculation-point where a processor is assigned, ROSES determines the processor's new ready time and inserts that processor into its proper place within the previously existing list-in-order-of-ready-time. If it happens that the new ready time equals the ready time of some previously existing sublist (even a sublist of membership 1), ROSES uses this rule: A processor having a newly assigned zero-length task keeps

its first place (i.e., is inserted immediately preceding the existing sublist), but a processor given any other assignment is placed immediately following the existing sublist.

ROSES uses the information of R and PR to construct and traverse the search tree. For example, ROSES uses the information of tready and pfwd to identify the pair $[t, p]$ characterizing the next search-tree vertex.

The structures R and PR store dynamic data. Suppose that ROSES decides to traverse a particular branch $[t, p, e]$ stemming from the current vertex. Then, as part of that traversal, ROSES updates evalue(p) to e. Suppose that ROSES then decides to extend the schedule yet further. That decision, together with the just-finished traversal of the branch $[t, p, e]$, may warrant a new value of tready(p) for the just-assigned processor p. Furthermore, it may warrant a change in pfwd(p) and pbak(p), to indicate a new position of p within the list that is arranged in nondecreasing order of tready(p'). Finally, it may warrant a change in the value of p, "the current processor."

Formally. the data structure R comprises a series of 4-word blocks. Below we first display the structure, and then review R's properties in "structural" terms.

$$\begin{aligned} \mathrm{R} = \{&[\mathrm{tready}(p'),\ \mathrm{evalue}(p'),\ \mathrm{pfwd}(p'),\ \mathrm{pbak}(p')] \\ &\text{for } p' = 0,1,\ldots,(N_p + 1)\}. \end{aligned} \tag{14}$$

Except for the cases $p' = 0$ and $p' = (N_p + 1)$, each block of R pertains to a real processor-index p'. Within each such real-processor block the first word stores the processor's future "ready time": i.e., the time at which the processor will finish its assignment and so be ready for reassignment. Of course, if a processor is assigned to perform a task ($E > 0$) then that processor's ready time is set equal to the e-starting time plus $L(e)$. If a processor p' is assigned to idle ($e = 0$), then its ready time is set equal to the earliest time — call it t_{ET} — that any task which is currently running (on another processor) will be finished. This time t_{ET} is of significance to the idling processor because t_{ET} is the earliest time when further tasks may become newly ready for assignment (ready, by virtue of their precedence requirements becoming fulfilled). Within each real-processor block of R the second word, evalue(p'), is the e-value assigned to processor p'. The third and fourth words, $pfwd(p')$ and $pbak(p')$ are linking pointers which define the list mentioned in Eqs. (10–13); i.e., the list arranging N_p processors in order of nondecreasing ready times.

The $p' = 0$ block of R is not a "real-processor" block; it is used only for forward-pointing to the processor that should be considered for assignment first — i.e., first among the N_p processors for which there are currently listed ready times. That first-to-be-considered processor has a ready time equal to the earliest finishing time, among the N_p known finishing times for $p' = 1, 2, \ldots, N_p$. However, some other processors may have that same ready time. Similarly, the $p' = (N_p + 1)$ block of R is not a real-processor block; it is used only for backward-pointing to that processor for which reassignment should be considered last: i.e., last among the N_p assignments for which there are currently forecasted times. That last-to-be- considered processor — call it p_{LAST} — has a ready time equal to the latest

finishing time for any currently running task. Here again, some other processors may have that same ready time.

Not only does $pfwd(0)$ point to the current processor p and not only does $pbak(N_p + 1)$ point to p_{LAST}; it is also true that $pbak(p)$ points to $p' = 0$, and $pfwd(p_{LAST}$ points to $p' = (N_p + 1)$. Thus, the pointers pbak and pfwd, together with the other elements in (14), define R as a doubly linked list of $(N_p + 2)$ blocks.

When R is viewed as a doubly linked list, its "real-processor" blocks are arranged in nondecreasing order of ready time tready(p'). When R is viewed as a set of Fortran arrays, its real-processor blocks are arranged in order of increasing processor index p'. From either viewpoint, the structure R begins with its $p' = 0$ block and ends with its $p' = (N_p + 1)$ block.

Finally we discuss how a schedule-under-construction is affected by the linked sequence of processors p' within a set of R-blocks having equal ready times. The linked sequence of p' indices within such an equal-ready-time set influences the *spt*-sequence of processor indices p' assigned within the schedule. However, that linked sequence of p' indices has no effect at all on the *spt*-sequence of e-values assigned within the schedule. The reasons for this lack-of-effect are that: (i) all tasks ready at the same *spt* are given the same opportunity to be selected (on the basis of their relative favor), and (ii) the readiness of a tasks at *spt* depends upon the time at which its predecessor-task subset becomes completed and the *spt*-index at which its predecessor-tasks were assigned, but not upon the *spt*-index at which its predecessor-task subset becomes completed. This matter is discussed further in Appendix A and Section IV.

III.7.3. The e-Alternatives Structure EA

The e-alternatives structure EA describes the branches stemming from each vertex along the search-tree path corresponding to the current schedule.

In brief: The task-branches stemming from successive vertices are described by a long list of e-values. The long list, called A, is used as a stack — one stack-section for each search-tree vertex *spt* along the path. In addition to the long list A there is a stack of pointer-triads — one triad for each search-tree vertex *spt* along the path. In each stacked pointer-triad, the first and third pointers indicate the beginning and end of the *spt* stack- section within the long list A. The second pointer indicates which branch is to be traversed next, in case ROSES visits vertex *spt* (perhaps after backtracking there) and finds that conditions warrant forward traversal from that vertex. Though A itself lists only task-branches, the second pointer can indicate either a particular task-branch, or the idling branch, or the condition that no satisfactory untraversed branches exist.

In more detail:

- The structure EA has two main substructures. Together, these two substructures define a stack of lists: one list for each vertex *spt* = 1, 2, 3, ..., spt_c in the current schedule-path. Each *spt*-tagged list contains, in decreasing order of f, L favor, all the task- indices $e > 0$ characterizing the branches stemming from search-tree vertex *spt*. It is understood that, in addition to these listed

task-branches, there is always one unlisted branch — the idling branch, which is most unfavored of all.

- Furthermore, for each search-tree vertex *spt* in the current schedule-path, the two main EA substructures together indicate whether there are any possibly-satisfactory as-yet-untraversed stemming-out branches; and if so, which among them is the most favored. That "most favored possibly-satisfactory as-yet-untraversed branch" is the one to be traversed next, in case ROSES is at the vertex *spt* and finds no reason to consider vertex *spt* unsatisfactory. (Vertex *spt* might be unsatisfactory in t or in D; see Section III.6.)

Thus: For the current vertex spt_c, the structure EA stores information concerning whether and how to extend the current (spt_c-1)-member schedule. For each previous vertex $\text{spt}' < \text{spt}_c$ the structure EA stores analogous information that will be helpful in case ROSES backtracks to that previous vertex spt'.

Fortran Expressions of the Substructures in EA

Next we define Fortran variables holding the information of the substructures in EA. Then we shall use these Fortran variables to review and extend our description of EA.

Suppose that a schedule-path extends from spt = 1 to spt = spt_c, and suppose also that the branches stemming from vertex spt_c have been determined. Then one of the two main EA substructures holds information expressible in terms of the Fortran array

$$\text{A(ia) for ia} = 1, 2, 3, \ldots, \text{LASTiaPLUSone}(\text{spt}_c), \tag{15}$$

which has elements storing task-indices e. The exhibited upper limit, LASTiaPLUSone(spt_c), is a datum included among those given by the second main substructure of EA. That second main substructure holds information expressible in terms of the Fortran pointers

$$\begin{aligned}
\text{iaFIRST} &= \text{FIRSTia(spt)} && \text{for spt} = 1, 2, \ldots, \text{spt}_c; && (16)\\
\text{iaCURRENT} &= \text{CURRENTia(spt)} && \text{for spt} = 1, 2, \ldots, \text{spt}_c; && (17)\\
\text{iaLASTplusONE} &= \text{LASTiaPLUSone(spt)} && \text{for spt} = 1, 2, \ldots, \text{spt}_c. && (18)
\end{aligned}$$

Each of (16), (17), (18) shows a scalar as well as an array-element. The three explicitly indexed pointer-arrays FIRSTia(spt), CURRENTia(spt), and LASTiaPLUSone(spt) by themselves indicate the pointer structure. However, for a given spt, it is often convenient to use the dynamic scalar iaFIRST instead of FIRSTia(spt); and similarly for iaCURRENT and iaLASTplusONE.

The precise meanings of A(ia), and of the pointers, are as follows:

At each search-tree vertex spt, the number of stemming-out task-branches is

$$\begin{aligned}
\text{nTASKbranches(spt)} &= [\text{LASTiaPLUSone(spt)} - \text{FIRSTia(spt)}];\\
&= \text{iaLASTplusONE} - \text{iaFIRST}\ .
\end{aligned} \tag{19}$$

If this number is nonzero, then the associated task indices $e > 0$ are given by the list

$$\text{A(FIRSTia(spt))}, \ldots, \text{A(LASTiaPLUSone(spt))}, \tag{20a}$$

or equivalently

$$\text{A(iaFIRST)} ,\ldots, \text{A(iaLASTplusONE–1)} \tag{20b}$$

where in (20b), iaFIRST and iaLASTplusONE refer to spt as in (16–18). In the special case that iaFIRST = iaLASTplusONE, Eq. (19) shows that the number of task-branches is zero. In that case the list for spt is considered empty. The end-element A(iaLASTplusONE) is not a member of the task-branch list. In fact, the value of A(iaLASTplusONE) is never used; only the pointer-value itself, iaLASTplusONE, is used. (The role of this value will be discussed further, shortly.)

As to the pointer iaCURRENT: If and only if iaCURRENT < iaLASTplusONE, the value of A(iaCURRENT) is the index of a possibly satisfactory untraversed task branch stemming from vertex spt. In that case iaCURRENT indicates the e-index of the next branch to be traversed if the search-procedure is at vertex spt and conditions warrant forward search from that point.

Implications of the Relative Values of the Pointers

For a given *spt*, the relative values of the pointers FIRSTia(spt), CURRENTia(spt), LASTiaPLUSone(*spt*) indicate the existence or nonexistence of certain kinds of branches stemming out from the vertex. One example has already been noted: If and only if iaCURRENT < iaLASTplusONE does there exist, stemming from the vertex *spt*, a possibly satisfactory as-yet-untraversed task-branch. This if–then condition involving iaCURRENT, and other such relative-value conditions, affect the decisions that ROSES makes about how to traverse branches stemming from the vertex *spt* and how to construct branches at vertex (*spt* + 1). Table 2 summarizes the main properties and implications of the pointers' relative values for a single search-tree vertex *spt*. Several of the conditions shown in Table 2 are used within if–then statements in the computer-code implementation of the ROSES methodology.

Other Properties of EA

After a search-tree part (vertex plus stemming-out branches) is constructed, the corresponding defining pointers FiRSTia(*spt*) and LASTiaPlUSone(*spt*) remain unaltered as long as that vertex remains in the current search-tree path. In contrast, the pointer CURRENTia(*spt*) undergoes revision every time ROSES traverses a branch from that vertex *spt*. For a newly constructed vertex *spt*, CURRENTia(*spt*) is initialized at FIRSTia(*spt*). Then after each traversal, CURRENTia(*spt*) is incremented appropriately so that — barring cases of an "unsatisfactory" vertex — the *spt*-tagged branches will be traversed successively, left to right (in decreasing order of favor), on successive visits to the vertex.

In some cases the list of e-choices at vertex $spt + 1$) is identical to a sublist within the list for the preceding vertex spt. In such a case the appropriate part of the spt section of A(ia) is "re-used" for vertex ($spt + 1$). That is, the three pointers associated with ($spt + 1$) are set so that they point to elements within the same physical A-section that was used for spt. Thus, the array A(ia) may not stack a physically separate list for each search-tree vertex. Nevertheless, the effect of the pointers is to define a unique list for each different search-tree vertex in the schedule-path, and all of the stated relations such as those in Table 2 hold.

Table 2. Relative Values of the e-Alternatives Pointers for a Single Search-Tree Vertex

Condition			Comment
iaFIRST	$\leq$	iaLASTplusONE	always true;
iaFIRST	$<$	iaLASTplusONE	iff there is at least one task branch;
iaFIRST	$=$	iaLASTplusONE	iff only an idling branch exists.
iaFIRST	$\leq$	iaCURRENT	always true;
iaFIRST	$=$	iaCURRENT	iff vertex spt is being visited for the first time;
iaCURRENT	$\leq$	iaLASTplusONE + 1	always true;
iaCURRENT	$<$	iaLASTplusONE	iff there exists at least one possibly-satisfactory untraversed branch, and the most favored such branch is a task-branch;
iaCURRENT	$=$	iaLASTplusONE	iff there exists at least one possibly-satisfactory untraversed branch, and the most favored such branch is the idling branch;
iaCURRENT	$>$	iaLASTplusONE	iff there are no more possibly-satisfactory untraversed branches;
iaCURRENT	$=$	iaLASTplusONE + 1	iff iaCURRENT > iaLASTplusONE

For further discussion of the structure EA, we refer to Appendix C. It discusses:

- The relation between the set of task-branches stemming from vertex ($spt - 1$) and the set stemming from vertex spt; and
- More details about the arrangement of the e-indices of these two sets of task-branches, within the long list A of the structure EA.

III.7.4. The Dynamic Task-Graph Structure DTG

The dynamic task-graph structure DTG is very plain; it involves no pointers or stacks or queues. We include it here simply because it is pertinent to the last

part of Section IV (Overview of the Code) and it needs more than just a few-line definition.

As noted earlier (see Section III.5), a task-graph vertex is "ready" if and only if it has no unfinished fan-in edges. The dynamic task-graph structure DTG includes information describing, for each task-graph vertex, how many fan-in edges are unfinished. Thus, the dynamic task-graph structure DTG relates (i) the fan-in-edge properties of the initial task graph with (ii) the traversed-branch history of the current schedule-path in the search tree.

Two substructures of DTG are called nfieus(v) and nfieuts(v). We pronounce them "en-fuse" and "en-futes"; and their names are mnemonic, as will be seen. Each of them describes the number of unfinished fan-in edges, but they differ in the detailed meaning of *unfinished.* Just below we give preliminary definitions (to be followed by more detailed explanation):

$$[\text{nfieus}(v) \text{ at } spt'] = \text{number of fan-in edges unfinished at spt', for task-graph vertex } v; \tag{21a}$$

$$[\text{nfieuts}(v) \text{ at spt}'] = \text{number of fan-in edges either unfinished at } t_{spt} \text{ or not assigned before spt', for task-graph vertex } v. \tag{21b}$$

The above definitions are formally correct, though they may need more explanation for a detailed understanding. However, even if one does not analyze or understand them in detail, they have these uses: They indicate that ROSES stores unfinished-edge information, and they indicate that there are complicating distinctions to be made in connection with the fact that t is nondecreasing with *spt* rather than increasing with *spt.* The more detailed explanation, presented just below, is included mainly for use in conjunction with reading the outline of routine Create in Section IV.5.6 ahead.

We begin with some terminology and notation. As usual, the word "edge" implies "task" ($e > 0$). Also as usual, the current schedule-path extends from search-tree vertex $spt = 1$ to search-tree vertex spt_c. Let subscripts s and ϕ indicate "start" and "finish," respectively. At search-tree vertex spt_s, edge e is assigned to start at time $t_s(e)$; it will finish at time $t_\phi \equiv t_s + L(e)$. Furthermore: If, within the current schedule, e ever plays the role of a finishing edge e_ϕ, then the search-tree vertex at which this happens is characterized by $spt_\phi(e)$ and t_ϕ.

Using the foregoing terminology we present, for the current schedule-path, the following definitions which describe three different classes of "finished":

An edge e is "finished at-or-before spt_c" (22)
iff $\text{spt}_\phi(e) \leq \text{spt}_c$, i.e.,
iff $\text{spt}_\phi(e) \leq$ and $\text{spt}_s(e) < \text{spt}_c$.

An edge e is "finished at-or-before $< t_c, \text{spt}_c >$" (23)
iff $t_\phi(e) \leq t_c$ and $\text{spt}_s(e) < \text{spt}_c$.

An edge e is "finished at-or-before t_c" (24)
iff $t_\phi(e) \leq t_c$.

In each of the definitions (22) and (23) the last-written condition, $\text{spt}_s < \text{spt}_c$, merely specifies that the edge e has been assigned at some point within the current schedule. In our subsequent discussion, phrases like "satisfies (23)" will mean "satisfies the iff-conditions stated in definition (23)." As will be explained in the next few paragraphs: The edges satisfying (22) are a subset of those satisfying (23), which in turn are a subset of those satisfying (24).

In definition (22) the last-written condition, $\text{spt}_s < \text{spt}_c$, is formally superfluous because it follows automatically from the first condition plus the truism that $\text{spt}_s(e) < \text{spt}_\phi(e)$. However, we have included that last condition explicitly, in (22), in order to make the following fact more immediately obvious: If an edge satisfies definition (22), then it also satisfies definition (23). This is so because $\text{spt}_\phi \leq \text{spt}_c$ implies $t_\phi \leq t_c$. Thus, the edges satisfying (22) are a subset of those satisfying (23).

In definition (23) the words "finished at-or-before $< t_c$, $\text{spt}_c >$" could be replaced by "known at spt_c to be finished at-or-before t_c;" because requiring the condition $\text{spt}_s < \text{spt}_c$ means requiring that ROSES "knows" the finishing time $t_\phi(e)$.

The edges satisfying definition (22) finish at t_c and so satisfy (24). So do the edges satisfying (23). However, there may also be additional edges satisfying (24); for it could happen that, at some spt_s, beyond spt_c an edge e will be assigned that will turn out to finish at $t_\phi(e) = t_c$. The t_ϕ of that yet-to-be-assigned e is unknown to ROSES; and so, such an edge is not "known at spt_c to be finished at t_c." Thus, the edges satisfying (22) are a subset of those satisfying (24), and the edges satisfying (23) are a subset of those satisfying (24).

Because "finished at-or-before $< t_c, \text{spt}_c >$" means "known at spt_c to be finished at-or-before spt_c," the following definition is not a trivial statement: Any edge which is not "finished" is said to be "unfinished."

Next, just below, we repeat definition (6a) from the initial task-graph structure ITG, and then finally define nfieus(v) and nfieuts(v) in detail:

$$\text{nfie}(v) = \text{number of fan-in edges to task-graph vertex } v \quad (6a)$$

$$[\text{nfieus}(v) \text{ at } \text{spt}_c] = \text{number of fan-in edges (to } v\text{) which are unfinished at spt} \quad (25a)$$

$$[\text{nfieuts}(v) \text{ at } \text{spt}_c] = \text{number of fan-in edges (to } v\text{) which are unfinished at } < t_c, \text{spt}_c >. \quad (25b)$$

The last defined variable, nfieuts(v), is important for constructing the search tree. When nfieuts(v) at spt_c is zero, then the task-graph vertex v is known to be ready at t_c; and so, all the unstarted edges fanning out from v are known to be ready at t_c. The e-alternatives at spt_c – i.e., the e-choices characterizing the branches stemming from a search-tree vertex spt_c — are restricted to ready tasks. For detailed implementation, the following points are helpful:

- An edge is "ready at *spt*" if an only if (1) it has not been assigned before search-tree vertex *spt*, and (2) it is a fan-out edge from a task-graph vertex at which nfieuts(v) = 0.

- An edge is "newly ready at *spt*" if and only if it is a fan-out edge from a task-graph vertex v which has nfieuts(v) = 0 at search-tree vertex *spt* but had nfieuts(v) $\neq$ 0 at search-tree vertex (*spt* $-$ 1).

Thus, the e-alternatives at search-tree vertex spt_c are restricted to tasks which have their required-predecessor tasks finished at or before $< t_c, \text{spt}_c >$. Because of redundancy-avoiding procedures, there are also further restrictions in the case that t at spt_c is the same as t at $\text{spt}_c - 1$.

From the above described properties of (23) and (24), it is easy to show that

$$\begin{aligned}[\text{nfieuts}(v) \text{ at } \text{spt}_c] &\leq [\text{nfieus}(v) \text{ at } \text{spt}_c] \\ &\leq [\text{nfieus}(v) \text{ at } \text{spt}_c - 1].\end{aligned}$$

For further information relevant to the use of nfieuts(v) in constructing the search tree, see Section IV.5.6 (the outline of the coded routine Create) and see Appendix A (paragraph 3 under **Redundancy-Avoiding e-Sequencing Rule**).

The dynamic task-graph structure DTG also includes the delay function $D(e)$ discussed in Section III.6.

IV. OVERVIEW OF THE CODE

This section culminates with detailed outlines describing the coded implementation of several major ROSES algorithms. Specifically, these outlines describe the ROSES-code Main routine and five principal subroutines. All of Section III and Appendices A, B, and C constitute preparatory material for understanding these outlines, and the present section includes further preparatory material. Specifically: Section IV.1 notes and justifies this report's emphasis on forward search. Section IV.2 briefly reviews the overall ROSES procedure in terms of the work done during one pass through the principal control structure of MAIN. Section IV.3 describes some terminology and notation to be used in the outlines. Section IV.4 draws attention to several kinds of complications handled by ROSES. Section IV.5 describes the algorithms in terms of outlines of routines. As indicated earlier, some of this report's arrays, etc. differ in detail from those actually coded in ROSES. Section IV.3 and Appendix D describe how this report's variables and routine-names differ from the ROSES code's variables and array-names.

IV.1. PRELIMINARY REMARKS

Here we briefly discuss scope and emphasis, considering these major aspects of the ROSES code:

a. Input and initializations;

b. Forward movement in the search tree;

c. Heuristics;

d. Backtracking (through one or many search-tree vertices); and

e. Output.

In this report we treat all these aspects but emphasize (b). For example, the six outlines in Section IV.5 describe the principal algorithms controlling (b), forward movement in the search tree. The other listed aspects are described, throughout this report, mainly in terms of how they affect forward movement. Heuristics and backtracking deserve some special comments here. ***As to heuristics:*** These are used to sequence and limit the ROSES code's investigations of different trial schedules. Among the present heuristics used by ROSES, only some are described within this report. The described heuristics include all of those which affect the first complete schedule that is constructed by ROSES, for a given input problem. Other heuristics exist in ROSES, and yet more will be added. (The present multiplicity of heuristics, and the plans for continual addition and refinement of heuristics, are consistent with calling ROSES an "expert" scheduler.) ***As to backtracking and heuristics, together:*** The conditions that provoke backtracking are such that there is little or no backtracking while ROSES constructs its first complete schedule for a given input problem. Furthermore, the present ROSES heuristics have proved so good that, in the practical applications investigated thus far, ROSES has never found a second-or-later schedule that excels its first-found solution. Therefore we have considered forward movement the aspect to emphasize, in this report.

IV.2. A PASS THROUGH THE MAIN CONTROL STRUCTURE OF ROSES

Following a set of data-initializations, the MAIN routine executes the basic computation-control algorithm of ROSES. Roughly speaking, each pass through this control structure (or "loop") starts at a definite vertex of the search tree (the "beginning-of-pass" vertex), and then, depending on conditions found within the pass, does one and only one of these three things:

1. Advances fully to the next consecutive vertex. Makes preparations for advancing forward from that end-of-pass vertex.

2. Advances toward the next consecutive vertex, but decides that there is not enough promise to continue beyond that vertex. Therefore, returns to the beginning-of-pass vertex. (In this case the end-of-pass vertex is the same as the beginning-of-pass vertex.)

3. Decides that advancing from the beginning-of-pass vertex is not worth doing. Therefore, backtracks from the beginning-of-pass vertex. This backtracking is done along the search-tree path corresponding to the current schedule. Once started, the pass's backtracking proceeds continuously until a vertex is reached where the e-assignment was other than a zero-length task.

IV.3. TERMINOLOGY AND NOTATION

- **Meaning of "Pass"**

In the rest of this report, "pass" always means a single execution of the basic control structure (or "loop") in the MAIN routine.

- **Meaning of "Earlier," "Later," Etc.**

This report's discussion often involves comparative terms like "earlier" and "later." We have tried to make clear, in each case, the kind of time or sequence meant: e.g., sequence of "wall-clock" times during a ROSES calculation, or sequence of times t in the triplets $[t, p, e]$ of a schedule, or sequence of placement within the spt-tagged list of a constructed schedule, or sequence of placement within the linked list of the processor- readiness structure PR. These sequences are not necessarily the same. For example,

- During its overall search, ROSES may construct many different schedules (by moving forward and backward in the search tree). Therefore neither t nor spt is generally monotonic with increasing ROSES execution-time.

- Within a single schedule, the series of spt-tagged triplets $[t, p, e]$ may include some groups of consecutive members having different spt but identical t. Therefore "earlier spt" is not the same as "earlier t."

- The task-finishings have consequences for new-task readiness, and the linked list of the processor-readiness structure PR could be interpreted as implying

a definite spt-sequence for task-finishings (even within a set in which all task-finishing times are the same). However, ROSES does not restrict its recognition of readiness consequences to the same spt- sequence as is implied for task finishings by PR's linked list.

(That is: Suppose that ROSES is considering triplet assignment at spt_a and will not assign processor p_x there because several other processors precede p_x in the linked PR list. Suppose, however, that p_x is listed as finishing task e_x at tready(p_x), and suppose also that the finishing of e_x completes the requirements for e_z. Then at spt_a ROSES may assign e_z even though ROSES will insist on delaying, until past spt_a, the assignment of the processor p_x whose completion of task e_x makes e_z assignable.)

Thus, to make clear what is meant by "earlier" or "later" in particular cases, we may write phrases like "a schedule that ROSES has constructed earlier during its overall search." However, it is often the case that shorter phrasing will suffice. If the reader keeps in mind the variety of meanings that "earlier" and "later" may have, then short phrases (such as "later- constructed schedule") should suffice to indicate which meaning is the intended one.

- **Suffixes BOP and EOP to Mean "Beginning-of-Pass" and "End-of-Pass"**

In ROSES, most of the principal data structures are dynamic. In many cases there is just one important updating of the structure within each forward-moving pass; and that updating occurs midway in the pass (not at the very beginning, not at the very end). The updating of different variables proceeds sequentially in ROSES computation time, not concurrently. Furthermore, the updating of one variable generally requires non-trivial usage of other variables. Thus, within a single pass there may be important use of a variable while it still has its beginning-of-pass value, and then important use after it has been updated from its beginning-of-pass value. To read the code (or an outline) with proper understanding, it is crucial to know the update-stage of each dynamic variable in each instruction.

Therefore we often show the update stage of a variable by appending, immediately after the variable's basic name, the letters BOP or EOP. For example, in the case of spt we write sptBOP or sptEOP. The ending BOP indicates the variable when it has its beginning-of-pass value; EOP indicates the variable when it has its end-of-pass value. (The meanings here are very strict. Thus for BOP, "beginning" means "the very beginning" not just "toward the beginning". For EOP, the phrase "end-of-pass value" means the value identical to that at the very end of the pass.) Occasionally we append the letters INT, to indicate versions which are intermediate — i.e., not guaranteed to have the beginning-of-pass value, but not guaranteed to have the end-of-pass value either.

Another set of examples, more complex than the pair sptBOP and sptEOP, consists of the pointers

CURRENTiaBOP(sptBOP),
CURRENTiaEOP(sptBOP),
CURRENTiaEOP(sptEOP),

all three of which play important roles, within a given pass. Such lengthened names are not graceful-looking. However, we tolerate them because they make it so easy for us to connote, without extensive discussion, the update stage of the variable.

In fact, these additions BOP, EOP, INT are nothing more than built-in *comments* to the variable names: for, in the case of any variable-name so terminated, the code ROSES may be assumed to incorporate statements such as the Fortran specifications

```
EQUIVALENCE (spt,          sptBOP,          sptEOP,          sptINT)
EQUIVALENCE (CURRENTia,    CURRENTiaBOP     CURRENTiaEOP     CURRENTiaINT)   (26)
```

Clearly then, the commenting device BOP, EOP, INT causes no wasting of storage space. Furthermore, each EOP value within one pass automatically becomes the BOP value for the next pass.

The fact that the endings BOP (etc.) are comments has other implications too. Note, for example, that sptBOP is *not* simply a saved value which may be used after spt has been updated. Instead, the very appearance of sptBOP means that spt has not yet been updated within the current pass. The ROSES code does occasionally need saved values-before-updating, to use together with values- after-updating. In such cases, truly separate variables are set. For example, ROSES sets sold = sptBOP early in a pass. Then later in that pass (past the point where updating to sptEOP has taken place), ROSES investigates whether or not sptEOP = sold. It would not be reasonable to investigate whether sptEOP = sptBOP, because the two are Fortran-equivalent. The only kind of instruction that properly includes both sptEOP and sptBOP is an updating instruction (e.g., sptEOP = sptBOP + 1).

- **Differences Between This Report's Variables and ROSES Code Variables**

This report uses some variables which differ mildly from those in the actual ROSES code. In particular: This report uses some names meant to be more mnemonic than their counterparts in ROSES. Also, this report uses some indexed arrays which are trivially simple transformations of ROSES-code indexed arrays. This is done in an attempt to allow explanation of all ROSES procedures in terms of indexing schemes which are very easy to discuss.

These mild deviations from the ROSES code should pose no problem for readers who may later deal directly with the code, for there is documentation available which describes the relation between this report's variables and ROSES counterparts. (See Appendix D and the following).

The three variables in $[t, p, e]$ deserve special comment. We shall first point out which ROSES-code variables are not to be identified with $[t, p, e]$ and then, which are to be so identified.

In the ROSES code there is a simple three-part structure [tsched(spt), psched(spt), esched(spt)] that is related to $[t, p, e]$; it will appear in our Sections IV.5.3 and IV.5.6 — the outlines of "CheckE" and "Create." However, the relation between

[tsched(spt), psched(spt), esched(spt)] and $[t, p, e]$ is not close enough for us to identify the two structures with each other. The two structures correspond with this exception: [tsched(spt), psched(spt), esched(spt)] is not defined until about midway in the pass; and moreover, it is defined only if ROSES decides to try continuing the current schedule beyond its spt-tagged triplet. In contrast, the triplet $[t, p, e]$ is investigated by ROSES quite early in the pass — earlier than the setting of [tsched(spt), psched(spt), esched(spt)].

In the program-outlines of Section IV.5, the proper counterparts to t, p and e are tready(p), p, and e.

For p, our earlier remarks about BCP and EOP apply straightforwardly. That is, there are no unusual problems concerning pBOP and pEOP.

The counterpart to tBOP is treadyBOP(pBOP), and the counterpart to tEOP is treadyEOP(pEOP). In addition, there is treadyEOP(pBOP). Furthermore, the ROSES code uses these scalars: told as a saved value of treadyBOP(pBOP), tNEXTinSCHED as the value of treadyEOP(pEOP), and tNEXTfcp (meaning "$t$next for the current processor") as the value of treadyEOP(pEOP).

As to e: The ROSES code does have a variable named e having the same meaning as the e in this report's triplet $[t, p, e]$. However, at the very beginning of a pass, the implicit value of e does not make a proper triplet with tBOP and pBOP. In fact, in some passes e is found to be "nonexistent"; i.e., there are no as-yet-untraversed branches at the search-tree vertex. ROSES does not set or use e, within a pass, until e is found to be "existent." After the first explicit setting of e within a pass, e is never revised within that pass. (Therefore it could be written as eEOP.) Although the ROSES e is not a beginning-of-pass value, it is does have a straightforward relation to beginning-of-pass variables. That relation is

$$e = \begin{cases} \text{Abop((CURRENTiaBOP(sptBOP))} & \text{if iaCURRENTbop} < \text{iaLASTplusONEbop} \\ 0 & \text{if iaCURRENTbop} = \text{iaLASTplusONEbop} \\ \text{undefined} & \text{if iaCURRENTbop} > \text{iaLASTplusONEbop} \end{cases} \quad (27)$$

where the suffix "bop" has the same meaning as "BOP," and where iaCURRENTbop has the same value as CURRENTiaBOP(sptBOP), and iaLASTplusONEbop has the same value as LASTiaPlusoneBOP(sptBOP).

IV.4. AVOIDING REDUNDANCIES

The program outlines in Section IV.5 will indicate in detail how ROSES treats:

zero-length tasks,

idling assignments,

e-assignments which begin simultaneously with each other,

e-assignments which finish simultaneously with each other, and

schedules which (if constructed) would be essentially redundant with each other.

These matters are all "special cases," in the sense that handling them requires elaborations of the basic ROSES constraint-satisfying and heuristic procedures. The elaborations are generally to avoid redundancies. Although we have called these matters "special cases," they do in fact occur often within applications of interest. One reason for the occurrence of simultaneous times is that the finishing of one task e_ϕ can make many tasks simultaneously ready. (This happens in cases where e_ϕ was the last unfinished edge to a task-graph vertex with many fan-out edges.) Another reason for the occurrence of simultaneous times is that message-passing tasks are sometimes treated as zero-length edges. The major reasons for encountering possibilities of redundant schedules are: the sameness of all processors, the assumed equidistance of processors, and the occurrence of simultaneous times.

The above-listed complications are interrelated, and so is their handling. The program-outlines in Section IV.5 describe the mechanics of *how* these special cases are handled. What we would like to indicate, also, is *why* the described program steps are correct for (1) exploring the search tree as fully as the ROSES heuristics warrant, while (2) avoiding the construction of essentially redundant schedules. In an effort to make this clear, we have included pertinent explanations in Section III, in Appendices A and B, and in comments within the program outlines.

The following two remarks may help readers to use the program outlines to understand the how and why of handling zero-length tasks and the other listed complications:

1. Because the listed complications are strongly interrelated, their handling is not confined to one or a few ROSES routines. (This diffusion is somewhat analogous to the pervasiveness of backtracking considerations.) To illustrate the pervasiveness of complication handling — i.e., redundancy avoidance — we list, below, some examples of complication handling described in the six outlines presented in the next section:

   - The algorithm of Main includes a backtracking procedure affected by zero-length tasks (see STEP 7).

   - The "CheckE" algorithm calculates nonzero lengths for idling assignments (see STEP 2).

   - The steps in "ForwAE" and "Create" are obviously dominated by provisions for treating *e*-assignments that have simultaneous finishing-times.

2. There are alternative schemes which could have been chosen, for modifying the basic ROSES procedures so as to accommodate the special complications. Because the possible choices are non-unique, and because parts of the chosen schemes are interrelated and are distributed among many programs (see comment 1), readers are likely to find that the instructions in any one program are insufficient to indicate what scheme has been chosen. Therefore examination of several programs, and some iterative reading, may be needed in order to develop a satisfying picture of the complication handling methods. This report's preparatory descriptions, in the main text and in Appendices A and B, should help to speed understanding.

IV.5. OUTLINES OF SELECTED ROUTINES

The present ROSES code incorporates over 20 routines. Six of the principal routines are outlined below. These outlines show the principal algorithms controlling forward motion in the search tree.

Unless otherwise indicated, readers may assume that all the mentioned variables are global (i.e., in Fortran common blocks available to all routines).

IV.5.1. Main

Purpose: Initialize variables in preparation for the main iterative control structure of ROSES — hereafter called the "main loop" of ROSES. Then execute this main loop.

Comment: The main loop constructs many schedules, triplet by triplet. To do this, the main loop moves forward and backward in the many-branched search tree. The directions and extents of these moves are determined by results calculated within each loop-pass. Although the main loop starts with a control instruction involving *spt*, the phrase "loop over *spt*" is inadequate as a capsule description. It is inadequate because the same ordinal number *spt* tags so many different vertices and branches in various sections of the search tree. A better description, indicating how each pass differs from others, is this: Each pass deals with (a) a particular search-tree vertex; and either (b.1) a particular as-yet-untraversed branch stemming from that vertex, or else (b.2) the knowledge that no such untraversed branch exists at that vertex. An alternative description (not using search-tree terms) is as follows: Each pass deals with a particular, unique combination of

a. *spt*BOP; plus all the other attributes describing a particular beginning-of-pass schedule: i.e., a partially competed schedule that comprises triplets $[t, p, e]$ tagged with selection-point indices 1, 2, 3, ..., (*spt*BOP − 1);

and furthermore,

b. a set of three beginning-of-pass pointers — *FIRSTia*BOP(*spt*BOP), *CURRENTia*BOP(*spt*BOP), *LASTiaPLUSone*BOP(*spt*BOP) — and the corresponding (pointed-to) elements of *A*bop. Together, these imply either (b.1) or else (b.2): viz.,

   b.1. a particular "current" triplet $[t, p, e]$ which, when added to the beginning-of-pass schedule, would produce an *spt*BOP-member schedule that satisfies the input-problem constraints and that is different from any *spt*BOP-member schedule previously constructed by ROSES.

   or else

   b.2. the knowledge that no such triplet exists.

In (b.2.), backtracking is necessary before forward movement should be made. In (b.l.), forward movement is appropriate. However, the newly formed, *spt*BOP-member schedule may be complete or not, and if complete, it may be worth continuing or not. Those matters will be determined and decided within the pass. Then appropriate preparations will be made, so as to prepare for the next pass.

Steps of Main:

1. Read input, and set static data closely related to input. To do this,

   1.1. Call SYSINT to set static data other than the task-graph description. For example, read in and set: the number of processors NP, the heuristic "delay"-limit $DMAX$, several file indices, and several values for end-of-list indicators.

   1.2. Call INGRAF to set the static task-graph structure ITG, the static array $nfie(v)$ describing the number of fan-in edges for each task-graph vertex, and other static task-graph data.

2. Initialize other data variables. To do this,

   2.1. Call LAGRAF to compute $f(edge)$, the static array describing the critical task-graph path length beginning with each edge.

   2.2. Call INITPR to initialize the dynamic processor-readiness structure PR.

   2.3. Call INITST to initialize dynamic data including: the e-alternatives structure EA, the selection-point index *spt*, and various time-bound variables such as *trequired* (which is the currently known lower bound for the **TIME_to_GOAL** of any feasible completed form of the current schedule-under-construction).

   2.4. Call INITQ to initialize data structures designed to save calculated values of time-bound variables, correlated with characteristics of the incomplete schedules for which these time-bound variables were calculated.

3. DO WHILE *spt*BOP exceeds 0
   (i.e., while there remain any as-yet-untried schedules worth investigating):

   3.1. Define the scalar pointers

$$\begin{aligned} iaFIRSTbp &= FIRSTiaBOP(sptBOP) \\ iaCURRENTbp &= CURRENTiaBOP(sptBOP) \\ iaLASTPLUSONEbp &= LASTiaPLUSoneBOP(sptBOP). \end{aligned}$$

   3.2. Determine whether current evidence indicates that it is worthwhile to investigate as-yet untried branches stemming from the beginning-of-pass vertex. To do this,

   - Check whether the pointers of step 4 identify a real triplet $[t, p, e]$ — as opposed to indicating that there is no available triplet which would extend the current schedule in a constraint-satisfying way.

   - Check (using brief tests comparing time-dimensioned variables calculated during the preceding pass) that there are no conditions which are common to all as-yet-uninvestigated branches stemming from the beginning-of-pass vertex, and which preclude a TIME_to_GOAL that

is satisfactorily short compared with the best one found earlier in this run of ROSES.

3.3. If the checks under 3.2 indicate that the proposed schedule-extension is indeed worth investigating, then

- Call FORWRD to investigate the extended schedule, to generate output if the goal is reached, and to revise dynamic data in preparation for later passes.

3.4. OTHERWISE (i.e., if either of the checks under 3.2 indicates that the current beginning-of-pass schedule is unworthy of continuation):

Call Bakwrd N times.

A fuller description of the preceding statement follows. Backtrack along the current schedule's search-tree path, at least as far as the preceding vertex (it is tagged $spt\text{BOP} - 1$); and then backtrack further, if necessary, to reach a vertex having $t < t(spt\text{BOP})$ — i.e., to reach a vertex at which the selected e was not a zero-length task. Do this backtracking by making N calls to Bakwrd. The integer N is essentially determined by Bakwrd. Bakwrd will roll back most dynamic variables to the values they had at the beginning of the last-previous pass in which the beginning-of-pass *spt* was equal to ($-N+$ the present pass's sptBOP). However, the Bakwrd routine will refrain from rolling back some of the e-alternatives variables (so as to prevent re-investigation of already-investigated schedules), and will refrain from rolling back some of the time-bound variables (so as to prevent construction of schedules which are known in advance to be unpromising for culminating in solutions better than previously found solutions).

3.5. End the if-block that began at step 3.3.

4. End the do-loop that began at step 3.

5. End the routine Main.

IV.5.2. Forwrd — Called from Main

Purpose: Check whether the current search-tree vertex is "satisfactory in D" (see Section III.6) and, if satisfactory, investigate the sptBOP-member schedule formed by adding the current triplet $[t, p, e]$ to the beginning-of-pass schedule.

More specifically: If the current search-tree vertex is satisfactory in D, then form the current sptBOP-member schedule and check whether it seems worth extending further. If it does, update variables so as to prepare for that extension. In any case, update variables to prepare for possible later return, by backtracking, to the beginning-of-pass search-tree vertex. If in the course of the investigation calculations are made which produce time-bound information of possible later interest, then save that time-bound information. (Some of these updates will serve also to inform the calling program, Main, what to do in its next pass.) If in the course of investigation, a goal-reaching solution of interest is found, then generate appropriate output.

Comment: Forwrd is a managing program. It effects most of its results by calling other subroutines to perform the detailed work.

Steps of Forwrd:

1. Check for excessively delayed tasks. To do this:

   Check whether *FIRSTia*BOP(*spt*BOP) points to a real edge; i.e., check whether the most favored (leftmost) branch at the current search-tree vertex is a task-branch. If so, then call that leftmost branch's task-index "*elong*" and check whether, if the current (*spt*BOP–1)-member schedule were augmented by the current candidate triplet, the resulting schedule would have $D(elong)$ exceeding $DMAX$. Such a $D(elong)$ implies that the task "*elong*" would be excessively delayed, within any schedule incorporating the current candidate-branch.

   In that excessive-delay case, not only is the current candidate-branch considered too unpromising to investigate, but all of the as-yet-untraversed branches stemming from the vertex are considered too unpromising to investigate. Therefore in that case, mark the vertex as having no as-yet-untraversed branches worth traversing. To do this, update the current-branch pointer *CURRENTia(spt)* from *CURRENTia*BOP(*spt*BOP) to

   $$CURRENTAia\text{EOP}(spt\text{BOP}) = LASTiaPLUSone\text{BOP}(spt\text{BOP}) + 1.$$

   Then return to Main. One immediate effect will be that Main will end the current pass without doing much other updating — e.g., will end the pass with *spt*EOP the same as *spt*BOP — and then will spend the next pass backtracking.

   If step 1 has not triggered return to Main, then continue as follows:

2. Revise the current-branch pointer *CURRENTia(spt)* from *CURRENTia*BOP (*spt*BOP) $\equiv$ *iaCURRENT*bp to

   $$CURRENTia\text{INT}(spt\text{BOP}) = iaCURRENT\text{bp} + 1 \ .$$

   With one kind of exception (see step 5.2 of CheckE), there will be no further updating of *CURRENTia* for *spt* = *spt*BOP in this pass. Thus in most cases, the above-described revision, to get *CURRENTia*INT, is the same as updating to get *CURRENTia*EOP(*spt*BOP). The purpose of such updating is to prepare for any later return, by backtracking, to the search-tree vertex *spt*BOP.

3. Save the beginning-of-pass value *tready*BOP(*p*BOP) as the scalar *told*.

4. Call CheckE. CheckE will set *e* and will check whether there are any "*e*-dependent" reasons — i.e., reasons depending on this pass's special *e* — for deciding that the *spt*BOP-member schedule is not sufficiently promising to continue. (Main and Forward have already determined that there are no "*e-in*dependent" reasons for rejecting continuation of the current schedule — i.e., no reasons known to be the same for all as-yet-untried *e* at the search-tree vertex.) CheckE will also update some variables. In particular, CheckE will update parts of the processor-readiness structure R; the results will be *evalue*EOP(*p*BOP) and *tready*EOP(*p*BOP).

5. If CheckE returns information indicating that the *spt*BOP-member schedule is not worth continuing, then return to Main.

6. Otherwise (i.e., if CheckE returns information indicating that continuation seems worthwhile): Continue the process (started by CheckE) of updating variables in preparation for furthering the *spt*BOP-member schedule. To do this, execute steps 7 through 11:

7. Update *D(elong)*, incrementing it by 1 (to get its EOP version). After this updating, the array *D(el)* will characterize the schedule incorporating the next as-yet-untraversed branch stemming from the current search-tree vertex.

   For explanation of what *D(elong)* means, see step 1 of Forwrd. See also Section III.6.

8. Call ForwPR, to finish updating the process-readiness structure PR. In particular, ForwPR will update the pointer parts of the doubly linked list R from their BOP versions to their EOP versions. ForwPR will also update *p* from *p*BOP to *p*EOP.

9. Call ForwEA. ForwEA will revise the *A*–array and pointer-set substructures of the *e*-alternatives structure EA, updating them from their BOP versions to their EOP versions. In addition, ForwEA will update *trequired* to *trequired*EOP, where *trequired* = the known lower bound to *TIME_to_GOAL* for any completed schedule incorporating the current, partial *spt*BOP-member schedule. ForwEA will also call a subroutine, Create, that will (among other things) check whether the goal has been reached, and if so call a subroutine (OUTDAT) to print output.

10. Update to the array *nfieus*EOP(*v*); where

    nfieus(v) = the number of fan-in edges unfinished at *spt*

    for each task-graph vertex *v*.

This updating is designed to describe the unfinished-edge situation at the current pass's end-of-pass search-tree vertex. At that vertex, processor *p*EOP becomes ready for reassignment. Because *nfieus(v)* needs changing only if a task gets finished, the array *nfieus(v)* is updated here only if *evalue(p*EOP) exceeds 0. In that case, the updating is

$$nfieus\text{EOP}(h) = nfieus\text{BOP}(h) - 1 \ ,$$

where *h* is the head of the edge having index *evalue(p*EOP); that is, *h* = *head(evalue(p*EOP)).

11. Set *time*EOP(*spt*EOP) = *trequired*EOP. Both sides represent the lower limit to *TIME_to_GOAL* for any completed schedule incorporating the current, partial *spt*BOP-member schedule.

12. Return from Forwrd to Main.

4.5.3. CheckE – Called from Forwrd

Purpose: The purpose has two parts, (a) and (b).

a. Check properties depending on *e*, to determine whether any of the back-track-provoking conditions 0, 1, 2, 3 holds.

   Each of these four conditions implies that, by adding the current triplet [*t,p,e*] to the beginning-of-pass (*spt*BOP–1)-member schedule, one gets an *spt*BOP-member schedule that is too unpromising to continue.

b. Update selected variables so as to prepare for subsequent work of two kinds – work to be done later within the current pass, and work to be done within the next pass and yet-later passes.

Comment: When CheckE is called, Main and Forwrd have already established that there exists a particular real triplet [*t,p,e*] constituting the current candidate for addition to the current beginning-of-pass (*spt*BOP–1)-member schedule. In fact, Main and Forwrd have already performed preliminary checks on the satisfactoriness of that triplet, and have not found it to be unsatisfactory. However, these preliminary checks did not involve detailed properties of *e*. Now CheckE will make further checks, taking into consideration the particular *e* in [*t,p,e*].

Special Note: In this outline, the section on Steps departs slightly from the actually coded CheckE. In brief: Our "steps" outline separates some of the intertwined parts of the actually coded program.

More specifically: In the actual CheckE, there is an intertwining of program-parts that (a) check for conditions 0, 1, 2, 3; and (b) update variables needed within this pass's CheckE and/or needed later than this pass's CheckE. That intertwining of program-parts promotes effciency of execution, but makes the code harder to understand. In order to conmunicate the functions of CheckE more clearly, we have written an outline which ignores some of the intertwining.

Steps of CheckE:

1. Set *e*.

   This is not a matter of *selecting e*; that selection was done implicitly prior to entering CheckE. The present step merely sets *e* to the value that is already implicitly – and uniquely – determined by the BOP version of the *e*-alternatives structure EA. See Eq. (27) of Section 4.3.

   We interpret the setting of *e* as adding the current triplet [t,p,e] to the beginning-of-pass (*spt*BOP–1)-member schedule, so as to form an *spt*BOP-member schedule. After setting *e*. CheckE Proceeds to investigate the properties of such an *spt*BOP-member schedule.

2. Set the three scalars *tNEXTfcp*, *tNEXTinSCHED* and *pNEXTinSCHED*. These will be used within the current operation of CheckE, and within other subroutines later in this pass. The meanings are:

tNEXTfcp = time for next assignment of the current processor *p*BOP;

tNEXTinSCHED = time for next *spt* in the schedule;

pNEXTinSCHED = processor for next *spt* in the schedule.

The settings depend on the value of *e* as follows:

- **If e > 0 – that is, if a task has been assigned, set**

  tNEXTfcp = *told* + L(e), where *told* ≡ *tready*BOP(*p*BOP) and has been passed from Forwrd to CheckE;

  tNEXTinSCHED = min [*tready*BOP(*pfwd*BOP(*p*BOP)), *tNEXTfcp*];

  pNEXTinSCHED = *pfwd*BOP(*p*BOP) if *tNEXTinSCHED* < *tNEXTfcp*

  pNEXTinSCHED =

  *p*BOP if *tNEXTinSCHED* = *tNEXTfcp*

- **If e > 0 – that is, if idling has been assigned, then set**

  tNEXTfcp = minimum of those times *tready*BOP(*p*') that exceed told;

  tNEXTinSCHED = *tready*BOP(*pfwd*BOP(*p*BOP));

  pNEXTinSCHED = *pfwd*(*p*BOP).

3. Check for condition 0 – the condition that the *e* of the current triplet [*t*,*p*,*e*] represents an edge having the same length and head as the edge in some other triplet previously added (as the *spt*BOP-tagged member) to the current beginning-of-pass schedule.

   In the foregoing sentence, "previously" means "in a previous pass, at a prior visit to the current search-tree vertex." The implication is that the current pass's visit to the search-tree vertex is a visit that has been reached by backtracking.

   Condition 0 is the redundancy condition described in Appendix B. When condition 0 holds, the current *spt*BOP member schedule has exactly the same *TIME_to_GOAL* possibilities as those of a schedule previously investigated by ROSES during this run.

4. Check, in sequence, for each of the other backtrack-provoking conditions – condition 1 (re idling); condition 2 (re critical path length), and condition 3 (involving dynamic programming).

   For additional description of these conditions, see further comments at the end of this program-outline.

5. In conjunction with making the checks of step 4, update some data to be used in later passes. Specifically:

   5.1 If condition 1 (excessive idling) does not hold, then update the global timebound variable idle time .

   5.2 If condition 2 holds, then not only is the current *spt*BOP-member schedule unsatisfactory (re critical path), but the same unsatisfactory feature would plague all as-yet-unconstructed schedules incorporating the current pass's (*spt*BOP – 1)-member schedule. Therefore, revise the branch-pointer *CURRENTia(spt)* from the value it has had since it was set by the Forwrd step-2 instruction *CURRENTia*INT(*spt*BOP) = *iaCURRENTbp* + 1. Revise it by setting

$$CURRENTia\text{EOP}(spt\text{BOP}) = iaLASTplusONEbp + 1 \ ,$$

This new setting of *CURRENTia* will forestall construction, in any later pass, of any further schedules incorporating the current pass's (*spt*BOP–1)-member schedule.

   5.3 Update the dynamic programming structures.

6. If steps 3 and 4 have shown that any of backtrack-provoking conditions 0, 1, 2, 3 holds: Set a flag to inform Forwrd that the newly constructed *spt*BOP-member schedule is not worth continuing, and return to Forwrd.

   This return to Forwrd leaves *spt*EOP = *spt*BOP. It will also result in many other variables (e.g., the processor-readiness structure) having the same EOP values as their BOP values. In effect, this return and its ramifications mean that, within the current pass of the main loop, ROSES has traveled forward along a particular search-tree branch stemming from the beginning-of-pass search-tree vertex, but then (still within the current pass) has backtracked to that same vertex in preparation for the next pass.

7. If none of the conditions 0, 1, 2, 3 holds, then the *spt*BOP-member schedule is considered worth continuing. In this case:

   7.1 Record the triplet [*t,p,e*] in a structure

$$[tsched(spt\text{BOP}),\ psched(spt\text{BOP}),\ esched(spt\text{BOP})].$$

   7.2 Similarly, record the finishing edge *evalue*BOP(*spt*BOP) as *efinishing(spt*BOP). This represents the edge which finished on processor *p*BOP at time *t* = *t*BOP = *tready*BOP(*p*BOP). It is the edge which, by finishing, freed *p*BOP to be the *p* of this pass's [*t,p,e*].

   7.3 Increment the selection-point index *spt* by setting

$$spt\text{EOP} = spt\text{BOP} + 1 \ .$$

   7.4 Update the non-pointer parts of the processor readiness quartet structure R. That is, set

$$evalue\text{EOP}(p\text{BOP}) = e,$$
$$tready\text{EOP}(P\text{BOP}) = tNEXTfcp.$$

(All other elements of *evalue* and *tready* remain constant throughout the pass.)

8. Return from CheckE to Forwrd.

Further Comments: Condition 1 indicates excessive idle time; its check is related to T_o (see Section III.6). Condition 2 indicates that there is an as-yet-unassigned edge with an *f(e)* that would be excessively long when added to *tNEXTinSCHED*. Condition 3 involves comparisons implemented by dynamic programming techniques. These three conditions may be described further in later documentation of ROSES.

IV.5.4. ForwPR – Called from Forwrd

Purpose: Complete the forward-updating (started in CheckE) of the processor-readiness structure PR.

More specifically: Update the pointer arrays *pfwd(pindex)* and *pbak(pindex)* of PR so as to move the *p*BOP-tagged block of PR to its proper new place within the doubly linked list of the PR structure. (The index *p*BOP identifies the processor recently assigned during this pass.) Also, update *p* from *p*BOP to *p*EOP, making *p*EOP identify the processor to be considered for assignment in the next pass.

Comment: The doubly linked list is to be arranged so that ready time is nondecreasing in the forward direction. Rearranging the list may be viewed as rearranging the order of processor indices *pindex*, or as rearranging the order of blocks within the ordered set

$$\{\ [\ tready(pindex),\ evalue(pindex),\ pfwd(pindex),\ pbak(pindex)\ ]:\ pindex = 0,\ 1,\ 2,\ \ldots\ ,\ N_p + 1\ \}\ .$$

Steps of ForwPR:

1. If *pNEXTinSCHED* (the next processor to be considered for assignment) is the same as *p*BOP (the processor assigned within the current pass), then no updating by ForwPR is necessary. In that case, return to Forwrd.

   (The data element *pNEXTinSCHED* is a parameter supplied by Forwrd to ForwPR. It is passed as a Fortran argument.)

2. Change the forward pointer in the *pindex*=0 block, and the backward pointer in the *pindex*=*pNEXTinSCHED* block, so as to identify *pNEXTinSCHED* as the first real processor in the doubly linked list.

   (Before this step, the processor was the second real processor in the doubly linked list. Therefore this step would be equivalent to removing the *p*BOP block from the doubly linked list, were it not for the fact that this step leaves the *p*BOP block with obsolete forward and backward pointers.)

3. Find the proper new place for *p*BOP in the doubly linked processor list. The new place should be such that, after *p*BOP is inserted there, the updated processor list will be in nondecreasing order of end-of-pass ready time *tready*EOP. (The array *tready*EOP(*pindex*) was set in CheckE.)

To execute this step, loop backwards through the doubly linked list, starting with *pindex*=$N_p + 1$ and successively considering the blocks that are pointed to by backward pointers. Here "considering a block" entails comparing its *tready(pindex)* with *tready*EOP(*p*BOP). In this way, find the place that just precedes the block (or set of blocks) having the smallest *tready* greater than *tready*EOP(*p*BOP). Such a place will *follow* any set of other blocks having *tready*=*tready*EOP(*p*BOP). Here the choice of "following" is an arbitrary convention. (Limiting to one definite place avoids some redundant schedules.)

4. Change four of the PR pointers so as to insert *p*BOP into its proper new place within the doubly linked list – i.e., into the place found in step 2.

   To execute this step: Update the forward and backward pointers in the *p*BOP block of PR. Also, update the backward pointer in the block that (according to step 2) should follow the *p*BOP block. Also, update the forward pointer in the block that (according to step 2) should precede the *p*BOP block.

5. Update the index *p* from *p*BOP to *p*EOP=*pNEXTinSCHED*, so that the updated index *p*BOP identifies the process to be considered for assignment in the next pass.

6. Return from ForwPR to Forwrd.

IV.5.5. ForwEA – Called from Forwrd

Purpose: Complete the forward-updating of the *e*-alternatives structure EA. Here "forward-updating" implies "for the case that *spt*EOP exceeds *spt*BOP by 1." The parts of EA to be updated are those parts relevant for the first visit to the search-tree vertex *spt*EOP.

Specifically, the parts to be updated are: that sublist of the array *A*eop which is directly related to *spt*EOP; and also, the pointers *FIRSTia*EOP(*spt*EOP), *CURRENTia*EOP(*spt*EOP), and *LASTiaPLUSone*EOP(*spt*EOP).

Comment: Earlier during this pass. Forwrd and CheckE revised EA to be suitable for subsequent revisits to this pass's vertex *spt*BOP. That preparation was done by updating from *CURRENTia*BOP(*spt*BOP) to *CURRENTia*EOP(*spt*BOP). Now ForwEA wil[prepare for the next pass by

   a. setting pointers *FIRSTia*EOP(*spt*EOP), *CURRENTia*EOP(*spt*EOP), and *LASTiaPLUSone*EOP(*spt*EOP), and

   b. setting, if needed, a physically new section within the *A*-array – a new section to which the three new pointers of (a) will refer.

Steps of ForwEA:

1. Check for the existence of conditions which make it unnecessary to construct (in preparation for *spt*EOP) a physically new sublist within *A*.

A physically new sublist is unnecessary when the proper sublist of A for *spt*EOP consists of the upper portion of the sublist used for *spt*BOP. Sufficient conditions for this "non-necessity" condition are as follows: (a) No tasks which were unready at the vertex <*spt*BOP,*told*> will be ready at the vertex <*spt*EOP,*tNEXTinSCHED*>; and (b) If the recently assigned e is a task, then it is permissible to limit the *spt*EOP sublist of A to edges e which are values less favored than the recently assigned e. Sufficient conditions for this are (see III.7.4 and Appendices A and C):

$$tNEXTinSCHED = told \text{ and } tNEXTfcp \neq told.$$

The last-written condition is equivalent to $L(e) \neq 0$, where is the assignment made earlier in this pass.

2. If the check of step 1 indicates that a physically new sublist is unnecessary, update only the EA pointer arrays and then return to Forwrd. To do this:

   2.1 First, set

   $$LASTiaPLUSone\text{EOP}(spt\text{EOP}) = iaLASTplusONEbp\ .$$

   2.2 Then if the recently assigned e was a task (i.e., if $iaCURRENTbp < ia\ LASTplusONEbp$): Start the new branch-set with the e-value that, in the *spt*BOP branch-set, was just rightward of the recently assigned e, and also, set the new current-branch pointer for the first visit to vertex *spt*EOP at that same just-rightward e-value. This means setting

   $$FIRSTia\text{EOP}(spt\text{EOP}) = iaLASTplusONE\text{bp},$$
   $$CURRENTia\text{EOP}(spt\text{EOP}) = iaLASTplusONE\text{bp}.$$

   2.3 However, if the assigned e was idling (i.e., if $iaCURRENT\text{bp} = iaLASTplusONE\text{bp}$), then restrict the branch-set at the vertex *spt*EOP to idling also. To do this, set

   $$FIRSTia\text{EOP}(sptEOP) = iaLASTplusONE\text{bp},$$
   $$CURRENTia\text{EOP}(spt\text{EOP}) = iaLASTplusONE\text{bp}.$$

   2.4 Then return to Main.

3. ELSE – i.e., if the check of step 1 does not show that a physically new sublist is unnecessary – then construct a physically new sublist for *spt*EOP within A. Also, set appropriate pointers (i) to associate the newly constructed sublist with *spt*EOP, and (ii) to indicate that the most favored (leftmost) branch is the branch to be traversed at the first ROSES visit to search-tree vertex EOP if that vertex is satisfactory. The physically new sublist for *spt*EOP is to be placed just beyond the *spt*BOP-list within A. In order to properly construct it, place it, and point to it:

   3.1 Set $FIRSTia\text{EOP}(spt\text{EOP})$ and $CURRENT\ ia\text{EOP}(spt\text{EOP})$ each to $LASTiaPLUSone\text{BOP}(spt\text{BOP})$.

3.2 Call Create. Among other things, Create will return an array "*buffer*," listing (in decreasing order of favor) tasks which are ready at *spt*EOP but which were not ready at *spt*BOP.

3.3 Find *sptPRIME*, as follows:

If $L(e) \neq 0$, then loop backward through the lists stacked in A, in order to find the smallest selection-point index such that *LASTiaPLUSone(sptPRIME)* = *LASTiaPLUSone*BOP(*spt*BOP). The sublist of A that is associated with this *sptPRIME* contains all the tasks ready at *tNEXTinSCHED*, except for those which will be in the array "*buffer*" of step 3.2. However, this sublist-associated-with-*sptPRIME* also contains some tasks which became assigned at selection-point indices equal or greater than *spt*EOP – 1 and therefore are not ready at *spt*EOP.

If $L(e) = 0$, where e is the most recent assignment, then set

$$sptPRIME = spt\text{EOP} - 1.$$

The sublist of A that is associated with this *sptPRIME* contains all the tasks ready at *tNEXTinSCHED*, except for those which are of favor exceeding the favor of e, and those which will be in the array *"buffer"* of step 3.2. However, this sublist-associated-with-*sptPRIME* also contains task e itself, which became assigned at *spt*EOP – 1 and so is not ready at *spt*EOP.

3.4 Construct a sublist-of-A for *spt*EOP by merging the newly ready edges in the list "*buffer*" (of step 3.2) with the edges within the sublist that is associated with *sptPRIME* of step 3.3; but omit any edges of that sublist-associated-with-*sptPRIME* which have already been assigned and so are not ready at *spt*EOP.

During this merging, keep all edges in decreasing order of f,L favor. (When an edge in the list-associated-with-*sptPRIME* has the same $f(e)$ and length as an edge in the "*buffer* list," then the edge in the *sptPRIME* list is considered more highly favored.)

3.5 From the length of the merged list in step 3.4, determine and set the pointer *LASTiaPLUSone*EOP(*spt*EOP).

4. Update *trequired*BOP to

$$trequired\text{EOP} = \max\,[trequired\text{BOP},\ tNEXTinSCHED + f(A\text{eop}(FIRSTia\text{EOP}(spt\text{EOP})))].$$

5. Return from ForwEA to Forwrd.

IV.5.6. Create – Called from ForwEA

Purpose: The purpose has two parts, (a) and (b).

a. Return, to ForwEA, an array *buffer(ib)*. This array should list, in decreasing order of f,L favor, all the newly ready edges – i.e., all the edges which are ready

at search-tree vertex *spt*EOP but which were unready at search-tree vertex *spt*BOP.

b. Check whether present information guarantees that the goal will be reached at $t_{(sptEOP)}$. If so: Record appropriate data describing the completion of the schedule; arrange for printing output; and decide whether to continue searching for additional schedules.

Comment: The logic of this program is complex if one considers in detail the handling of cases where zero-length tasks occur. For such cases, a fully explicit description of the reasoning behind this program would involve the detailed differences discussed in Section III.7.4: i.e., the differences between edges finishing at-or-before *spt*, edges finishing at-or-before $t_{(spt)}$, and edges finishing at-or-before $< t_{(spt)}, spt >$. In addition, a fully explicit description of this program's logic would involve explanations depending on the conditions under which Create is called: i.e., if *L(e)* is nonzero then *tNEXTinSCHED* is different from *told*. The outline below makes some compromises between brevity and explicitness; it describes some but not all of the details concerning what the program does, and some but not all of the details concerning why the described steps are appropriate for accomplishing the stated purposes.

Steps:

1. Find all the newly ready edges – i.e., edges which are ready at search-tree vertex *spt*EOP but which were not ready at search-tree vertex *spt*BOP. Place these newly ready edges, in decreasing order of favor, in an array "*buffer*." In order to do the aforementioned edge-finding and edge-placing:

   Find all the newly ready task-graph vertices – i.e., the vertices v_n which are ready at search-tree vertex *spt*EOP but which were not ready at search-tree vertex *spt*BOP. For each such newly ready vertex v_n, find each fan-out edge e_{ni} and then insert it into its proper place within an overall array "*buffer*" that will list all such edges e_{ni} from all such vertices v_n. In order to do all of the aforementioned v_n-finding and e_{ni}-finding and e_{ni}-placing:

   1.1 Set data that will remain constant during calculation of the array *buffer(ib)*. Examples are:

   head(e), where e = the e-value assigned earlier during this pass.

   e0flag. Set this to 1 if *L(e)* is zero, but to 0 if *L(e)* exceeds zero.

   1.2 Initialize data that will vary during calculation of the array *buffer(ib)*. Examples are:

   pindex, initialized to *p*EOP.

   nfieuts(v), initialized to *nfieus*BOP(v).

   The initial value given to *nfieuts* is not the proper value of *nfieuts(v)* at search-tree vertex *spt*EOP. However, for selected indices v, subroutine Create will revise the initial value so that *nfieuts(v)* will take on its proper value at search-tree vertex *spt*EOP.

The arrays *nfieuts* and *nfieus* were defined in section III.7.4. The array *nfieuts(v)* is an array internal to Create.

1.3 Loop over *pindex* using the sequence in which *pindex* is forward-linked in the doubly-linked processor-readiness list R. [That list R is forward-linked in order of nondecreasing *tready(pindex)*.] Treat all *pindex* in the first sublist of the linked structure R: i.e., all *pindex* in the sublist within which *tready(pindex)* = *tNEXTinSCHED*.

The variable *tNEXTinSCHED* was calculated in CheckE, returned to Forwrd, and passed by Forwrd through ForwEA to Create.

To execute this loop over *pindex*:

- For selected task-graph vertices, revise *nfieuts(v)* to find its proper value at the search-tree vertex *spt*EOP.

  The selected group comprises those task-graph vertices v such that *nfieuts(v)* is different at *spt*EOP than it was at *spt*BOP; because it is only for such task-graph vertices that a zero *nfieuts(v)* implies a newly ready vertex. In order for *nfieuts(v)* to be different at *spt*EOP than it was at *spt*BOP, v must be a vertex such that either (i) $t_{(spt\text{EOP})}$ exceeds $t_{(spt\text{BOP})}$ and there are some edges, assigned before *spt*EOP, which have v as a head and which finish at $t_{spt\text{EOP})}$, or else (ii) $t_{(spt\text{EOP})} = t_{(spt\text{BOP})}$ and v is the head of a zero-length task beginning at *spt*BOP.

  Therefore to make the selected revision:

  For each *pindex*, determine the associated task-graph vertex v which is the head of *evalue(pindex)*. Decrement the transient value of *nfieuts(v)* by 1 for each edge $e'' = evalue(pindex)$ satisfying the following two conditions: e'' is a task rather than idling, and if $L(e) = 0$ then $head(e'') = head(e)$.

- For each revised element *nfieuts(v)* that becomes zero, the vertex v is newly ready at *spt*EOP. Therefore for each *pindex* such that the *nfieuts(v)* is revised and is seen to be 0, find the edges fanning out from v and insert them, in decreasing order of favor, into the array "*buffer*."

  The edges fanning out from v are found by using the *ITG* substructures *G* and *hfoe(v)*. The order of favor is found by using the arrays *f(edge)* and *L(edge)*.

- Then end the loop on *pindex*

2. Check whether present data guarantee that the goal will be reached at $t_{(spt\text{EOP})}$. This is so if *nfieuts(goalnode)* = 0.

3. If the check of step 2 shows that the goal will be reached at $t_{(spt\text{EOP})}$, then do not bother to make and record all the further assignments (at *spt*EOP and beyond) which would formally reach the goal through idling-branch extensions of the partially completed search-tree path that goes up to vertex *spt*EOP. Instead,

proceed immediately to record information describing the completed schedule, to prepare for continued searching, and to make the decision as to whether to continue searching. In detail:

3.1 Set *TIME_to_GOAL = tNEXTinSCHED.*

3.2 Set the stacked block [*tsched(spt*EOP),*psched(spt*EOP),*esched(spt*EOP)] to [*TIME_to_GOAL,* 0, – 1].

3.3 Call OUTDAT to print output.

3.4 Set the variable *INITIAL* to False, to indicate that at least one solution (i.e. one complete schedule) has been calculated.

3.5 Set *time2undercut = TIME_to_GOAL - tbetter,* where *tbetter* is an input indicating how much improvement is demanded of later-constructed schedules, in order for ROSES to to consider them worth constructing.

3.6 Set *MaxIdle Time* = the maximum idle time (summed over all processors) allowable, in any yet-to-be-constructed schedule, if that yet-to-be constructed schedule is to have a *TIME_to_GOAL* shorter than *time2undercut.*

This means that *MaxIdleTime* will equal *(NP*time2undercut - Tzero),* where *Tzero* is the lower bound T_o defined in Section III.6.

3.7 Decide whether to cease or to continue searching for an improved solution. If the decision is to cease, then sept *spt=9* (because a value *spt=0*, when finally returned to Main, will cause the main loop to end). To do this deciding and zeroing:

- If $time2undercut \leq MinTime$, or if $MaxIdleTime \leq 0$, then set *spt = 0.*

  Here *MinTime* is a lower-bound-on-"the-achievable" that has been calculated by a subroutine of CheckE in the course of checking for Condition 3.

4. Return to ForwEA.

V. AN IMPLEMENTATION EXPERIMENT: INVERSE DYNAMICS EQUATIONS FOR A ROBOT MANIPULATOR

Let us now address some of the practical considerations related to the implementation of ROSES-scheduled algorithms on the NCUBE[5] concurrent computer. To fix the ideas, we consider a robot-related calculation: the solution of the inverse dynamics equations of a manipulator arm. This problem was chosen for illustrative purposes, i.e., because it is relatively "simple," while exhibiting the nonlocal communication and structural irregularity characteristics of interest. Below we start with rather general discussions of robot-arm equations, and parallel algorithms for solving them. Then we specialize to ROSES and to the NCUBE in discussing the ROSES-scheduling of a particular robot-arm algorithm, GOLEM, and the implementation of GOLEM on the NCUBE.

V.1. NEWTON-EULER INVERSE DYNAMICS

Several state-of-the-art formalisms are currently available to efficiently solve the inverse dynamics problem of a serial link manipulator, in which forces or torques are predicted based on desired motion. In the Newton-Euler formalism the equations for each link are written in link-fixed coordinate systems in order to simplify the calculation of the inertia tensors. A set of recurrence relations allows the angular velocities, angular accelerations and linear accelerations at the center-of-mass of each link to be successively calculated from the base to the end effector. Net forces and torques acting on each link's center-of-mass are then obtained. Forces and torques acting at the joints are subsequently calculated in a recursion from the "hand" to the base. Joint actuator torques or forces are determined from a knowledge of the orientation of each joint. The detailed derivations can be found in Luh and Lin's seminal paper.[26] A display of the Newton-Euler equations of motion is included in Table 3, with notation as follows:

w_i	=	angular velocity of link i
$\dot{w}_i$	=	angular acceleration of link i
$\ddot{p}_i$	=	linear acceleration of link i
F_i	=	net force acting on link i
Γ_i	=	net torque acting on link i about the center-of-the-mass
p_i^*	=	origin of the i-th coordinate system with respect to the (i-1)th
r_i^*	=	position of center of mass of link i with respect to the origin of link i
J_i	=	inertia tensor about center of mass of link i
A_i^{i-1}	=	frame transformation matrix from (i-1)th to i-th coordinate systems (denoted as A_i^- in the table); by analogy, A_i^+ will denote the frame transformation A_i^{i+1}
f_i	=	force exerted on link i by link i-1
γ_i	=	moment exerted by link i-1 on link i

The symbols w_i and w have identical meanings. Lightface type rather than boldface has been used, even for vector quantities and matrices.

Table 3. Robot Inverse Dynamics Computational Tasks as Characterized by their Principal Equations (Newton-Euler Formalism, Six-Link Manipulator)†

Task #	Rotational Link	Prismatic Link
1–6	$\mathrm{w}_i = A_i^-(w_{i-1} + z_{i-1}\,\dot{\mathrm{q}}_i)$	$w_i = A_i^- w_{i-1}$
7–12	$\dot{\mathrm{w}}_i = A_i^-(\dot{\mathrm{w}}_{i-1} + z_{i-1}\ddot{\mathrm{q}}_i + w_{i-1} \times z_{i-1}\dot{\mathrm{q}}_i)$	$\dot{\mathrm{w}}_i = A_i^- \dot{\mathrm{w}}_{i-1}$
13–18	$V_i^{(1)} = w_i \times (w_i \times p_i^*)$	$V_i^{(1)} = w_i \times (2A_i^- z_{i-1}\dot{\mathrm{q}}_i + w_i \times p_i^*)$
19–24	$\ddot{\mathrm{p}}{-}i = A_i^-\ \ddot{\mathrm{p}}_{i-1} + \dot{\mathrm{w}}_i \times p_i^* + V_i^{(1)}$	$\ddot{\mathrm{p}}_i = V_i^{(1)} + A_i^-(\ddot{\mathrm{p}}_{i-1} + z_{i-1}\ddot{\mathrm{q}}_i) + \dot{\mathrm{w}}_i \times p_i^*$
25–30	$V_i^{(2)} = w_i \times [w_i \times r_i^*]$	
30–36	$F_i = m_i[V_i^{(2)} + \dot{\mathrm{w}}_i \times r_i^* + \ddot{\mathrm{p}}_i]$	
37–42	$\Gamma_i = J_i\dot{\mathrm{w}}_i + w_i \times (J_i w_i)$	
43–48	$V_i^{(3)} = \Gamma_i + (P_i^* + r_i^*) \times F_i$	
49–54	$f_i = F_i + A_i^+ f_{i+1}$	
55-60	$V_i^{(4)} = p_i^* \times (f_i - F_i)$	
60–66	$\gamma_i = A_i^+\gamma_{i+1} + V_i^{(3)} + V_i^{(4)}$	

†$\dot{\mathrm{q}}_i$ and $\ddot{\mathrm{q}}_i$ are the first and second time derivatives of generalized coordinates. The joint index i runs from 1 to 6 for tasks 1 to 48, and from 6 to 1 for tasks 49 to 66.

The symbols z represent unit vectors. Typical initial conditions are $w_o = \dot{w}_o \equiv 0; \ddot{p}_o = gz_o$, thereby absorbing gravity into the initial acceleration to reflect the simplified form of the force balance equation

$$f_i = F_i + A_i^+ f_{i+1} \quad . \tag{28}$$

Initial conditions for the backward recursion are obtained from the specification of f_{N+1} and γ_{N+1}, the external force and moment exerted on the hand. The joint actuator torques or forces are simply (omitting friction):

$$\tau_i = \bar{\gamma}_i \cdot A_i^- z_{i-1} \quad or \quad \phi_i = f_i \cdot A_i^- z_{i-1} \quad . \tag{29}$$

V.2. PARALLEL ALGORITHMS FOR INVERSE DYNAMICS

The pioneering work of Luh and Lin on scheduling of parallel computations for a computer controlled mechanical manipulator[26] has established a solid foundation for further research and development of parallel algorithms for robot dynamics. In their approach, one CPU is associated with each joint (or link). Each CPU is connected both to a primary memory, which stores local programs and data, and to a "common" memory, located between adjacent CPUs, which stores common data and information necessary for interprocessor communication. The Newton-Euler formalism provides the computational framework. Because of the dynamic coupling between adjacent links, Luh and Lin developed a parallel formulation of the inverse dynamics problem in terms of a multicomputer task-scheduling optimization problem under series-parallel precedence constraints. Their solution, based on a generalization of the branch-and-bound algorithm, exhibits, however, several significant limitations. Most importantly, the bijective computer-to-link mapping is not ideal for real-world applications, since it is subject to single-point failures. Furthermore, because of the underlying topology, the system suffers from severe load unbalance, i.e., some processors are very underutilized. Finally, the issues of intertask communication and synchronization are not directly addressed.

In recent years, there has been an increased interest in the development of parallel algorithms for inverse robot dynamics.[26–30] As reported by Kasahara and Narita,[38] it seems that most of these studies do not involve an implementation on an actual multiprocessor system. Results are therefore often presented in terms of "number of additions and multiplications" and their theoretical equivalent of processor clock cycles,[37] ignoring many fundamental constraints of multiprocessing such as communication overheads or saturation effect bottlenecks. Furthermore, the emphasis is in general on architectures fully dedicated to a specific algorithm formulation.

Our approach is to use a single large-scale multiprocessor system, the NCUBE, for the concurrent solution of all major algorithms involved in the operation of the autonomous robot. Since the NCUBE is reconfigurable (i.e., subcubes of specified dimension can be allocated in real time), this approach attempts to make the best possible use of the available computing resources.

V.3. ROSES

Now we discuss input to ROSES, and output from ROSES, for the robot manipulator problem.

One of the first and most important steps in attempting to solve a problem on a concurrent computation ensemble is decomposing the problem into a set of computational tasks (or "processes") each to be performed on a single processor of the concurrent ensemble. For the robot-arm problem described above, our

proposed decomposition involves 66 computational tasks for a 6-degrees-of-freedom manipulator. The proposed decomposition is indicated in Table 3; it differs somewhat from the decomposition used by Luh and Lin.[26]

Problem decomposition induces precedence constraints among the computational tasks, and the distributed nature of a concurrent computational system translates those constraints into message-passing requirements. Such tasks and constraints are convenient to consider in terms of task graphs.

Task graphs in the parallel processing literature generally use nodes (vertices) to denote tasks, and edges to describe communication links or precedence constraints. However, we prefer a somewhat different model. In a full ROSES task graph, edges represent tasks (computational or communication), while nodes (vertices) represent synchronization points for precedence constraints. This was discussed in Section II.5, where we mentioned an input list including, not only all computational tasks, but also message-passing tasks.

In fact a user of ROSES may input either of the following:

[1] *A list including only computational tasks* – with each listed task accompanied by a sublist identifying that task's immediate predecessors in a precedence requirement description involving computational tasks only; or

[2] *A complete ROSES-task-graph edge list*, including not only all computational edges but also all message-passing edges – each one of these edges being accompanied by identification of its head-node and its tail-node (so that these nodes can be interpreted properly, as synchronization points for precedence constraints).

If the user inputs [1], then an optional "preparatory" part of ROSES needs [2]. An example of [1]-to-[2] conversion is indicated by Tables 4 and 5; it is for inverse dynamics and the Stanford arm.

Note that both Table 4, type [1] information, and Table 5, type [2] information, include task-describing algorithmic parameters such as Mul #, the number of multiplications in a task. Besides needing input that describes tasks, ROSES needs input that describes the concurrent ensemble on which the multi-task job is to be run. In particular, ROSES needs the total number of active processors (this may be input as a hypercube order), and the performance times for basic arithmetic operations.

The major, schedule-generation part of ROSES was described in Sections III–IV. It proceeds through a search to generate what we shall call a ROSES search-output schedule. For each processor in the ensemble, this schedule assigns a specific set of tasks. Furthermore, for each task the schedule stipulates a numerical starting time (relative to a zero-point time at which the entire multi-task job begins). Thus, the ROSES search-output schedule is very time-specific.

Table 4. Computational Tasks for the Solution of the Newton-Euler Robot Inverse Dynamics Equations As Input to ROSES*

Task ID	Mul #	Add #	Pr. Con. #	Prec.	IDS	Const.	Task ID	Mul #	Add #	Pr. Con. #	Prec.	IDS	Const.
1	8	6	1	0			34	9	9	3	10	22	28
2	8	6	1	1			35	9	9	3	11	23	29
3	8	5	1	2			36	9	9	3	12	24	30
4	8	6	1	3			37	12	6	2	1	7	
5	8	6	1	4			38	12	6	2	2	8	
6	8	6	1	5			39	12	6	2	3	9	
7	10	8	1	0			40	12	6	2	4	10	
8	10	8	2	1	7		41	12	6	2	5	11	
9	8	5	1	8			42	12	6	2	6	12	
10	10	8	2	3	9		43	6	9	2	31	37	
11	10	8	2	4	10		44	6	9	2	32	38	
12	10	8	2	5	11		45	6	9	2	33	39	
13	12	6	1	1			46	6	9	2	34	40	
14	12	6	1	2			47	6	9	2	35	41	
15	15	8	1	3			48	6	9	2	36	42	
16	12	6	1	4			49	8	8	1	36		
17	12	6	1	5			50	8	8	2	35	49	
18	12	6	1	6			51	8	8	2	34	50	
19	14	14	2	7	13		52	8	8	2	33	51	
20	14	14	3	8	14	19	53	8	8	2	32	52	
21	14	15	3	9	15	20	54	8	8	2	31	53	
22	14	14	3	10	16	21	55	6	6	2	36	49	
23	14	14	3	11	17	22	56	6	6	2	35	50	
24	14	14	3	12	18	23	57	6	6	2	34	51	
25	12	6	1	1			58	6	6	2	33	52	
26	12	6	1	2			59	6	6	2	34	53	
27	12	6	1	3			60	6	6	2	31	54	
28	12	6	1	4			61	8	11	2	48	55	
29	12	6	1	5			62	8	11	3	47	56	61
30	12	6	1	6			63	8	11	3	46	57	62
31	9	9	3	7	19	25	64	8	11	3	45	58	63
32	9	9	3	8	20	26	65	8	11	3	44	59	64
33	9	9	3	9	21	27	66	8	11	3	43	60	65

*The column headiings have these meanings: Task ID is the same as Task # in Table 3. Mul # means number of multiplications. Add # means number of additions. Pr. Con. # means number of immediate predecessor computational tasks in a task graph limited to computational tasks. The columns headed Prec., IDS, and Const. show the indices ID of the immediate predecessor computational tasks.

Table 5. Robot Inverse Dynamics Task Graph
As Generated by Roses from the Information in Table 4.†

Nodes		Cost		Edge	Nodes		Cost		Edge
Head	Tail	Mul	Add	No.*	Head	Tail	Mul	Add	No.*
3	2	8	6	1	103	102	8	8	51
5	4	8	6	2	105	104	8	8	52
7	6	8	5	3	107	106	8	8	53
9	8	8	6	4	109	108	8	8	54
11	10	8	6	5	111	110	6	6	55
13	12	8	6	6	113	112	6	6	56
15	14	10	8	7	115	114	6	6	57
17	16	10	8	8	117	116	6	6	58
19	18	8	5	9	119	118	6	6	59
21	20	10	8	10	121	120	6	6	60
23	22	10	8	11	123	122	8	11	61
25	24	10	8	12	125	124	8	11	62
27	26	12	6	13	127	126	8	11	63
29	28	12	6	14	129	128	8	11	64
31	30	15	8	15	131	130	8	11	65
33	32	12	6	16	133	132	8	11	66
35	34	12	6	17	4	3	0	0	67
37	36	12	6	18	6	5	0	0	68
39	38	14	14	19	8	7	0	0	69
41	40	14	14	20	10	9	0	0	70
43	42	14	15	21	12	11	0	0	71
45	44	14	14	22	16	3	0	0	72
47	46	14	14	23	16	15	0	0	73
49	48	14	14	24	18	17	0	0	74
51	50	12	6	25	20	7	0	0	75
53	52	12	6	26	20	19	0	0	76
55	54	12	6	27	22	9	0	0	77
57	56	12	6	28	22	21	0	0	78
59	58	12	6	29	24	11	0	0	79
61	60	12	6	30	24	23	0	0	80
63	62	9	9	31	26	3	0	0	81
65	64	9	9	32	28	5	0	0	82
67	66	9	9	33	30	7	0	0	83
69	68	9	9	34	32	9	0	0	84
71	70	9	9	35	34	11	0	0	85
73	72	9	9	36	36	13	0	0	86
75	74	12	6	37	38	15	0	0	87
77	76	12	6	38	38	27	0	0	88
79	78	12	6	39	40	17	0	0	89
81	80	12	6	40	40	29	0	0	90
83	82	12	6	41	40	39	0	0	91
85	84	12	6	42	42	19	0	0	92
87	86	6	9	43	42	31	0	0	93
89	88	6	9	44	42	41	0	0	94
91	90	6	9	45	44	21	0	0	95
93	92	6	9	46	44	33	0	0	96
95	94	6	9	47	44	43	0	0	97
97	96	6	9	48	46	23	0	0	98
99	98	8	8	49	46	35	0	0	99
101	100	8	8	50	46	45	0	0	100

*The indices which are ≤ 66 refers to computational tasks, and correspond to the Task ID numbers in column 1 of Table 4. The indices > 66 refer to message-passing tasks.

Table 5, Cont'd

Nodes		Cost		Edge	Nodes		Cost		Edge
Head	Tail	Mul	Add	No.	Head	Tail	Mul	Add	No.
48	25	0	0	101	96	85	0	0	151
48	37	0	0	102	98	73	0	0	152
48	47	0	0	103	100	71	0	0	153
50	3	0	0	104	100	99	0	0	154
52	5	0	0	105	102	69	0	0	155
54	7	0	0	106	102	101	0	0	156
56	9	0	0	107	104	67	0	0	157
58	11	0	0	108	104	103	0	0	158
60	13	0	0	109	106	65	0	0	159
62	15	0	0	110	106	105	0	0	160
62	39	0	0	111	108	63	0	0	161
62	51	0	0	112	108	107	0	0	162
64	17	0	0	113	110	73	0	0	163
64	41	0	0	114	110	99	0	0	164
64	53	0	0	115	112	71	0	0	165
66	19	0	0	116	112	101	0	0	166
66	43	0	0	117	114	69	0	0	167
66	55	0	0	118	114	103	0	0	168
68	21	0	0	119	116	67	0	0	169
68	45	0	0	120	116	105	0	0	170
68	57	0	0	121	118	65	0	0	171
70	23	0	0	122	118	107	0	0	172
70	47	0	0	123	120	63	0	0	173
70	59	0	0	124	120	109	0	0	174
72	25	0	0	125	122	97	0	0	175
72	49	0	0	126	122	111	0	0	176
72	61	0	0	127	124	95	0	0	177
74	3	0	0	128	124	113	0	0	178
74	15	0	0	129	124	123	0	0	179
76	5	0	0	130	126	93	0	0	180
76	17	0	0	131	126	115	0	0	181
78	7	0	0	132	126	125	0	0	182
78	19	0	0	133	128	91	0	0	183
80	9	0	0	134	128	117	0	0	184
80	21	0	0	135	128	127	0	0	185
82	11	0	0	136	130	89	0	0	186
82	23	0	0	137	130	119	0	0	187
84	13	0	0	138	130	129	0	0	188
84	25	0	0	139	132	87	0	0	189
86	63	0	0	140	132	121	0	0	190
86	75	0	0	141	132	131	0	0	191
88	65	0	0	142	1	123	0	0	192
88	77	0	0	143	1	125	0	0	193
90	67	0	0	144	1	127	0	0	194
90	79	0	0	145	1	129	0	0	195
92	69	0	0	146	1	131	0	0	196
92	81	0	0	147	1	133	0	0	197
94	71	0	0	148					
94	83	0	0	149					
96	73	0	0	150					

After an acceptable ROSES search-output schedule has been generated, actual execution on a multiprocessor is in order. To facilitate such execution, we invoke an optional, concluding part of the ROSES; it generates "run-time" control information to be fed to the multiprocessor. This run-time control information includes the following basic features of the ROSES-optimized schedule:

for each processor,

- the set of computational tasks to be performed, and
- the sequence in which these tasks should be performed.

However, the run-time control information omits the following time specific feature of the ROSES search-output schedule: numerical starting times for the tasks.

The run-time control information is fed to the multiprocessor along with a routine for each computational task in the ROSES task graph. Also fed in are an appropriate host routine, and ordinary input data needed to complete the specification of the problem to be solved concurrently. Within the computational-task routines, there are write-message instructions and read-message instructions. A correlated write-read pair corresponds to a message-passing task in the ROSES task graph. Given the aforementioned run-time control information and routines and input data, the multiprocessor has enough information to fully satisfy the multi-task job's precedence constraints – as we next explain.

Precedence constraints relating tasks on the same processor are satisfied because the multiprocessor will comply with that part of the run-time control information which specifies a sequence for each subset of tasks on the same processor. However, there is no need to specify direct sequencing constraints relating tasks on different processors, because such precedence constraints are taken care of by the following blocking feature: *execution is disallowed past a read-message instruction, until a message having the requested message-characteristics has been received and read.* This blocking feature is part of the operating system on NCUBE processors. Our convention is that, except for delays caused by such blocking, each processor is to perform its sequence of task without interruption.

What about the matter of specifying proper message-characteristics – i.e., the matter of matching write-read partners properly so that blocking will assure satisfaction of the precedence constraints? The present ROSES code facilitates this by using the ROSES-optimized task-to-processors assignments to prepare run-time information that specifies for *each computational task*,

- the number of messages to be received, and for each such message, its originating processor and its "message type" (a word that further characterizes the message);
- the number of messages to be sent, and for each such message its destination processor and its "message type."

V.4. THE GOLEM CODE

The Newton-Euler inverse dynamics formalism has been implemented in a computer code named GOLEM. It has been programmed to utilize, as part of its input, a run-time control information file generated by ROSES. We have have attempted to avoid "tailoring" the code for a specific robot arm. In particular, there is no limit on the number of joints, their type (rotational or prismatic) or on the manipulator configuration that GOLEM can handle. All matrices, vectors and scalars involved in the solution of the inverse dynamics equations (see Table 1) are stored in two one-dimensional arrays, denoted BC and NC, for reals and integers respectively. Storage requirements are calculated at run time, pointers are defined, and memory is allocated dynamically.

In order to enhance potential interactions with other robot activities sharing the hypercube, the architecture of GOLEM is modular. Key stages in the computational flow are modular interface, dynamic memory allocation, database processing, model initialization (e.g., frame transformation matrices, inertia tensors), actual solution of the equations and display of the results. Two modes of operation are available. In the sequential mode all computations take place on the Intel 80286/80287 processors resident on the NCUBE Peripheral Subsystem (host). In the concurrent mode the equations are solved on the hypercube nodes (i.e., the single-chip processors in the NCUBE ensemble).

Let us now briefly examine the basic steps involved in setting up the concurrent computation. Several system functions are provided[39] that allow use of the ensemble by FORTRAN and C programs. All functions return an integer*4 result; negative values correspond to an error code. First the host allocates the requested hypercube:

$$ISH = NOPEN(\ NOH\ ) \tag{30}$$

In the above statement NOH is an integer*2 variable giving the hypercube order. The channel number to be used when referencing that hypercube is the integer*2 variable ISH. The second step involves the loading of programs onto the nodes of the previously allocated hypercube. One can load either the same program onto all nodes or a different program onto each node. The only restriction in the preliminary release of the NCUBE system is the limit of one program per node. Thus, it is currently the responsibility of the user to provide multitasking capabilities at the node level, if multitasking is required by his particular solution algorithm. This is the case for GOLEM. Our approach is to download the same program to all nodes, i.e., in principle each node could execute all tasks in Table 3. However, only the tasks scheduled to be run on a particular node will be activated. To achieve this capability, we had to develop a "focused addressing" algorithm which uses the appropriate switching and synchronization information provided by ROSES. The host's download statement has the form:

$$istat = NLOAD(\ ISH, codex, IN, WA, NWA\ ) \tag{31}$$

where IN is an integer*2 variable identifying the node that is to receive the program, and codex is a character string that contains the name of the file to be loaded. The user must provide a work buffer (WA) that is at least as large as the size of the

node program. The variable NWA gives the size of the working area in bytes. For large applications one must, of course, send different code segments to each node, but the focused addressing scheme is still valid.

The third step involves the sending of data to the nodes. For a particular node IN, this is accomplished by the host using the statements:

$$istat = NWRITE(\ ISH, BC, NLOR * 4, IN, mtypr\ ) \tag{32}$$

$$jstat = NWRITE(\ ISH, NC, NL01 * 4, IN, mtypi\ ). \tag{33}$$

Here the "message-type" parameters mtypr and mtypi are used simply to indicate whether the data in the message are reals or integers. We download the complete working memory (arrays BC and NC of run-time word-sizes NLOR and NLOl, respectively) to all nodes. This is consistent with the approach outlined above. Furthermore, the array pointers have been stored in NC, and can be selectively retrieved at each node for maximum flexibility. Clocks are now initialized and we proceed with the solution of the equations on the hypercube ensemble.

Finally, the host must receive the problem responses from the appropriate nodes. The preferred implementation is clearly via interrupts. (A possible alternative is to continuously test for any arriving message from any node using the system function NTEST.) The message is then retrieved with the statement

$$isize = NREAD(\ ISH, WB, NWB, IN, LL\ ) \tag{34}$$

where WB is a buffer of size NWB bytes. This buffer is assumed to be large enough to hold the longest message that one would expect to receive from any node. IN denotes the source (originating node) of the message read. The message type LL is defined as the pointer to the subarray in BC (or NC) where the incoming message needs to be stored. Since a positive value of "isize" represents the size in bytes of the received message, this is accomplished as follows (assuming again 4-byte words):

```
NML = isize/4
CALL VECEQV(BC(LL+1), WB, NML )
```
(35)

After all expected messages have been received, timing is performed, the hypercube is deallocated, and the final results are displayed.

The communication primitives used by the nodes are semantically and syntactically quite similar to the ones described above for the host peripheral subsystem, and we will not enter into details. The most significant differences involve the use of an additional primitive, WHOAMI, and (in GOLEM) the use made of the message-type arguments. The subroutine WHOAMI is called by each node program to establish the node's identity and its relationship to the host for the current hypercube partition:

```
CALL WHOAMI( IDN, IDP, IDH, NOH )
```
(36)

All arguments are integer*2 variables and denote, respectively, the identities of the node, the calling process and the host, and the hypercube order allocated to the problem. In GOLEM, the message-type parameters used for internode communication are defined as 16-bit "packed words." The six most significant bits represent a pointer position (not value) in array NC. The next 9 bits encode task identity information obtained from ROSES. The least significant bit is used to determine whether the data in the message are reals or integers.

V.5. PRELIMINARY RESULTS

We present now the timing results for our ROSES-scheduled inverse dynamics calculations. These results are preliminary in the sense that the CESAR NCUBE machine is a "beta-site" prototype. The "time score" of the solution should be expected to change for the commercial ("production") version of the NCUBE/ten (a ten-dimensional hypercube), due to probable changes in system performance characteristics such as processor clock rate, operational status of the instructions, or compiler optimization capabilities. Thus, rather than list absolute timings for the computation, we prefer to show here more general results, such as expected levels of processor load balance (throughput). The assignment of the tasks in Table 3 to nodes of a two-dimensional hypercube is given in Table 6 for the forward and backward recursions. The processor utilization results shown in Table 7 refer to the most unbalanced node and are given for two, three, and four processors. The improvement in performance observed (a factor of two in the non-asymptotic domain over previously reported results[26]) is particularly significant in light of the fact that the NCUBE's individual nodes are much more powerful (by at least an order of magnitude), which the load balance results do not explicitly reflect. This should open the possibility for real-time control of flexible manipulators, where many more degrees of freedom are involved.

The fundamental role of the task scheduler in achieving good concurrent computation efficiencies for irregular robotics problems of the type discussed above can not be overemphasized. Thus, it is important to provide well-defined benchmarks against which the performance of available codes can be tested. As a first step in that direction, we have carried out a comparison between the recent results of Kasahara and Narita,[38] the original work of Luh and Lin,[26] and ROSES. To provide a fair basis for the comparison, we now assume that each processor has the same performance parameters as the ones used by Luh and by Kasahara (40 μs for a floating point add, 50 μs for a floating point multiply, rather than the 2μs associated with the NCUBE processor design). Furthermore, instead of using the task partition of Table 3 and the cost estimates for the general form of the equations, we adopt the task partition and "specialized" costs given by Luh and also used by Kasahara. For completeness, their task data are reproduced here in Table 8. The results of the comparison are summarized in Table 9. The agreement between our ROSES results and the results of Kasahara and Narita is excellent. However, whereas their method requires mainframe computing power,[38] ROSES runs on an Intel 80286 (IBM PC-AT, or NCUBE Peripheral Subsystem). Furthermore, notwithstanding the fact that the current version of our code is an unoptimized prototype, the time required to schedule over 200 tasks (88 computational and 140 message-passing) was approximately 12s on a 6MHz IBM PC-AT.

Table 6. Task Assignment of the Inverse Dynamics Equations for a Two-Dimensional Hypercube*

Node 0	Node 1	Node 2	Node 3
1	7	2	25
13	8	14	3
19	9	4	15
20	10	16	11
21	5	28	6
22	17	12	18
23	29	30	41
24	42	27	26
36	40	34	39
49	33	46	45
55	32	35	38
61	31	47	44
62	37	48	
63	43	50	
64	51	56	
65	57	52	
66	53	58	
	59	54	
		60	

*In each column the entries correspond to the Task # entries in Table 3, and to the Task ID entries in Table 4.

Table 7. Processor Load Balance for the Solution of the Inverse Dynamics Equations of the Stanford Manipulator

Item	This Work				Luh & Lin[26]
Processors	1	2	3*	4**	6
Load†	1	.96	.95	.61	.37
Speed-up	1	1.96	2.87	3.22	2.56

*We allocate an order-2 cube, but schedule tasks on three nodes only.

**Asymptotic domain critical length reached: using more than 4 processors for this simple example wastes resources.

†Load factor for most unbalanced node in system.

Table 8. Benchmark Parameters for the Solution of the Newton-Euler Equations of a Robot Manipulator*†

Task ID	Mul #	Add #	Pr. Con. #	Prec.	IDS	Const.	Task ID	Mul #	Add #	Pr. Con. #	Prec.	IDS	Const.
1	0	0	1	0			45	10	4	1	41		
2	0	0	1	0			46	4	1	1	43		
3	2	0	1	1			47	3	6	3	44	45	46
4	1	0	1	1			48	3	0	1	43		
5	3	0	3	2	3	4	49	9	3	1	41		
6	1	0	1	1			50	0	3	2	48	49	
7	2	0	1	1			51	4	3	1	41		
8	1	0	1	1			52	2	0	1	41		
9	4	2	2	1	8		53	4	5	2	43	52	
10	2	0	1	2			54	4	2	1	44		
11	6	2	1	7			55	6	2	1	51		
12	2	0	1	9			56	2	0	1	53		
13	0	6	3	10	11	12	57	0	6	3	54	55	56
14	10	4	1	7			58	12	6	1	51		
15	4	1	1	9			59	6	3	1	53		
16	3	6	3	13	14	15	60	3	6	3	57	58	59
17	3	0	1	9			61	9	6	1	53		
18	9	3	1	7			62	15	9	1	51		
19	0	3	2	17	18		63	0	3	2	61	62	
20	0	0	2	7	9		64	6	3	1	60		
21	2	0	1	20			65	0	3	2	63	64	
22	9	2	1	20			66	4	5	1	47		
23	0	1	1	13			67	4	2	1	65		
24	0	6	3	21	22	23	68	4	1	1	47		
25	6	2	1	20			69	0	6	3	50	67	69
26	2	0	1	20			70	4	5	2	37	66	
27	3	6	3	24	25	26	71	4	2	1	69		
28	3	0	1	20			72	4	1	1	37		
29	9	3	1	20			73	0	6	3	40	71	72
30	0	3	2	28	29		74	4	5	2	27	70	
31	4	3	1	20			75	6	2	1	70		
32	2	0	1	20			76	4	4	2	73	75	
33	4	5	2	20	32		77	2	0	1	27		
34	4	2	1	24			78	0	6	4	30	74	76
35	10	4	1	31			79	0	3	2	16	74	
36	4	1	1	33			80	2	0	1	74		
37	3	6	3	34	35	36	81	0	2	2	78	80	
38	3	0	1	33			82	4	1	1	16		
39	9	3	1	31			83	0	6	3	19	81	82
40	0	3	2	38	39		84	4	5	2	5	79	
41	4	3	1	31			85	4	1	1	79		
42	2	0	1	31			86	4	4	2	83	85	
43	4	5	2	33	42		87	4	1	1	5		
44	4	2	1	34			88	0	6	3	6	86	87

*Adapted from Tables 7 and 8 of Ref. 26, where the equations were "specialized" to the Stanford 6-DOF manipulator.

†For the meanings of column headings, see footnote of Table 4.

Table 9. ROSES Benchmark Results – Scheduled Times for Solution of the Newton-Euler Inverse Dynamics Equations for the Standard 6-DOF Manipulator Using the Parameters of Table 8

Number of Processors	Time for Optimal Schedule Found: Luh & Lin[26] ms	Kasahara & Narita[30] ms	ROSES (this work) ms	Known Lower Bound* ms	ROSES Exec. Time s(IBM-AT)
1	24.80	24.83	24.80	24.80	12.99
2	–	12.42	12.43	12.40	12.81
3	–	8.43	8.49	8.43	12.52
4	–	6.59	6.67	6.38	12.30
5**	–	5.86	5.88	5.67	11.84
6	9.70	5.73	5.78	5.67	11.68
7	–	5.69	5.67	5.67	11.75
8	–	–	5.67	5.67	11.94

* See this report's Section III.6, especially paragraphs 2 through 5.
** Beginning of asymptotic domain: optimal schedule on any multiprocessor would equal the critical path of the graph.

VI. CONCURRENT COMPUTATON, MACHINE INTELLIGENCE AND ROBOTICS

In this concluding section we comment briefly on the potential implications, for robotics, of recent advances in concurrent computation. We concentrate our remarks on the areas of machine vision and manipulator dynamics, in which significant research efforts are underway in the Oak Ridge Laboratory's Center for Engineering Systems Advanced Research (CESAR).

VI.I. MACHINE PERCEPTION

The field of machine perception (e.g., vision, tactile sensing, etc.) has been, in particular over the last decade, closely associated with progress in computer architectures, as evidenced by the detailed bibliographies found in the literature.[40–42] One of the main objectives of this association was to help overcome severe drawbacks in artificial vision systems such as latency and propensity to fail except in simple structured domains.

In the past, the computational emphasis has essentially been on SIMD architectures for simple preprocessing tasks such as convolution and mask evaluation; pipelined architectures have been used for executing sequences such as blurring/differentiation/zero-crossing. To develop a realistic machine vision system[43] one needs to go beyond these retina-level tasks and include some of the higher level components of human perception. For example, humans seem to perceive effortlessly characteristics such as colinearity, direction, periodicity, coarseness and continuity. This suggests that a machine vision system ought to include efficient mathematical transformations that can handle these effects.

For such complex processing options, where the type of processing is influenced by the context of the image, MIMD concurrent computers are required. Despite the fact that the processing speed of neurons is slow by silicon chip standards, human perception is extremely fast. This implies that human perception achieves its speed through massive parallelism, involving perhaps billions of processing elements. The same will almost certainly be true of successful machine vision systems. To achieve this goal, the CESAR research program on Human Analog Vision has, from the outset, investigated algorithms that can be executed on concurrent computer architectures.[43] NCUBE multiprocessors of various dimensions have been used in this research.

Vl.2. ROBOT DYNAMICS

For many of the applications in which mechanical manipulators are used today, real-time performance can be achieved in a well structured environment, using quasi standard bus-based multi-microprocessor architcctures. Such manipulators generally consist of a base-anchored open-link articulated chain, composed of a few (typically six) rigid links connected by rotational or prismatic joints. The dynamic behavior of a manipulator is modeled, as shown in Section V, by a set of coupled, highly nonlinear equations of motion. The efficient solution of these equations is required, both for the design of advanced real-time control algorithms, as well as for carrying out "planning" activities, either at the machine intelligence level of the autonomous robot, or at the man-machine interface for telerobotic applications.

A major emphasis in the design of future robots will be structural flexibility[44] and joint compliance.[45] Such models will increase very significantly the complexity of the equations of motion and associated control algorithms. Furthermore, since such calculations need to be carried out in a common computational framework with other robotic activities including vision, sensor fusion, navigation, ... it is essential that the computers on board be able to operate in a concurrent fashion.

VI.3. FINAL REMARKS

The development of concurrent computers raises several challenging issues. How powerful should each processor be? How should the processors communicate with each other? How should the workload be divided among the processors? How does one make sure that processors are not sitting idle waiting for input from other processors? To address the fundamental computational problems underlying the development of machine intelligence and robotics, we can now start from one of the most promising advances in the field of computer science, i.e., VLSI-driven hypercube architectures for concurrent computation. To utilize these architectures properly we should have ready-to-use, efficient methods for optimizing the time-dependent assignment of tasks-to-processors – methods suitable for a wide range of hard real-time applications. ROSES is one contribution toward that goal.

REFERENCES

1. C. L. Seitz, "Concurrent VII Architectures," *E Trans. Comp.* **C33**, 1247 (1984).

2. C. L. Seitz, "The Cosmic Cube," *CACM* **28(1)**, 22 (1985).

3. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison Wesley, Reading, Massachusetts (1980).

4. P. Wilson and R. Gisburn, "IMS T424 Transputer," preliminary data sheet, Inmos Corporation, Colorado Springs, Colorado (August 1984 et seq.).

5. J. Barhen and J. F. Palmer, "The Hypercube in Robotics and Machine Intelligence," *Comp. Mech. Eng.* **CIME-5(4)**, 30 (1986).

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

7. E. G. Coffman, *Computer and Job Shop Scheduling Theory*, J. Wiley, New York (1976).

8. M. J. Gonzalez, "Deterministic Processor Scheduling," *Comp. Surv.* **9(3)**, 173 (1977).

9. R. L. Graham, E. L. Lawler, J. K. Lenstra and A. H. G. R. Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Annals Descrete Math.* **5**, 169 (1979).

10. J. T. Robinson, "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms,"*IEEE Trans. Soft. Eng.* **SE- (1), 24** (1979).

11. Z. Rosberg, "Process Scheduling in a Computer System," *IEEE Trans. Comp.* **C-36(7)**, 633 (1985).

12. C. V. Ramamoorthy, K. M. Chandy and M. I. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System."

13. E. B. Fernandez and B. Bussel, "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules," *IEEE Trans. Comp.* **C-22(8)**, 745 (1973).

14. M. R. Garey, R. L. Graham and D. S. Johnson, "Performance Guarantees for Scheduling Algorithms," *Oper. Res.* **26(1)**, 3 (1978).

15. S. C. Sarin and S. E. Elmaghrabi, "Bounds on the Performance of a Heuristic to Schedule Precedence-Related Jobs on Parallel Machines," *Int. J. Prod. Res.* **22(1)**, 17 (1984).

16. J. K. Lenstra, A. H. G. Rinnoy-Kan and P. Brucker, "Complexity of Machines Scheduling Problems," *Ann. Disc. Math.* **1**, 343 (1977).

17. J. D. Ullman, "NP-Complete Scheduling Problems," *J. Comp. Syst. Sci.* **10**, 384 (1975).

18. B. Simons and M. Sipser, "On Scheduling Unit-Length Jobs with Multiple Release-Time/Deadline Intervals," *Oper. Res.* **32(1)**, 80 (1984).

19. H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Soft. Eng.* **SE-3(1)**, 85 (1977).

20. S. H. Bokhari, "On the Mapping Problem," *Trans. Comp.* **C-3(3)**, 207 (1981).

21. C. L. Monma, "Linear-Time Algorithms for Scheduling on Parallel Processors," *Oper. Res.* **30(I)**, 116 (1982).

22. T. H. Lai and S. Sahni, "Nearly On-Line Scheduling of Multiprocessor Systems with Memories," J. of Algor. **4**, 353 (1983).

23. C. C. Price, "The Assignment of Computational Tasks Among Processors in a Distributed System," *Proc. AFIPS-National Computer Conf.*, pp. 291-296, Chicago, IL (May 1981).

24. J. A. B. Fortes and F. Parisi-Presicce, "Optimal Linear Schedules for the Parallel Execution of Algorithms," *Proc. IEEE 1984 International Conference on Parallel Processing*, pp. 322-329 (August 1984).

25. Y. C. Chow and W. H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *IEEE Trans. Comp.* **C-28**, 354 (1979).

26. J. Y. S. Luh and C. S. Lin, "Scheduling of Parallel Computation for a Computer Controlled Mechanical Manipulator," *IEEE Trans. Syst. Man. Cyb.* **SME-12**, 214 (1982).

27. H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. Comp.* **C-33**, 1023 (1984).

28. J. Barhen, "Robot Inverse Dynamics on a Concurrent Computation Ensemble," *Proc. 1985 International Computers in Engineering Conference,* Vol. 3, pp. 415-429, Boston, MA (August 1985).

29. B. Blake and K. Schwan, "A Fast Scheduling Algorithm for Real-Time Systems," preprint, Dept. of Comp. and Info. Sciences, The Ohio State University (September 23, 1985).

30. E. Dekel and S. Sahni, "Parallel Scheduling Algorithms," *Oper. Res.* **31(1)**, 24 (1983).

31. C. R. Weisbin, J. Barhen, G. de Saussure et al., "Machine Intelligence for Robotics Applications," *Proc. 1985 Conference on Intelligent Systems and Machines*, pp. 47-57, Oakland University, Rochester, MI (April 1985).

32. R. B. McGhee, D. Orin, D. R. Pugh et al., "A Hierarchically Structured System for Computer Control of a Hexapod Walking Machine," *5th IFTOMM Symposium on Robot and Manipulator Systems*, Udine, Italy (June 1984).

33. J. Barhen, C. R. Weisbin and G. de Saussure, "Real-Time Planning by an Intelligent Robot," *Proc. 1984 International Computers in Engineering Conference*, pp. 358-360, Las Vegas, NE (August 1984).

34. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA (1983).

35. P. Gupta, "Multiprocessing Improves Robotics Accuracy and Control," *Computer Design*, **21**, 169 (November 1982).

36. R. H. Lathrop, "Parallelism in Manipulator Dynamics," *Int. Jour. Rob. Res.*, 4, #2, 80 (1985).

37. R. Nigam and C. S. G. Lee, "A Multiprocessor-Based Controller for the Control of Mechanical Manipulators," *IEEE Jour. Rob. & Aut.,* RA-1, #4, 173 (1985).

38. H. Kasahara and S. Narita, "Parallel Processing of a Robot Arm Control Computation on a Multiprocessor System," *IEEE J. Rob. Aut.,* RA-1, #2, 104 (1985).

39. Caine, Farber and Gordon, Inc., *Fortran 77 Guide for Use on NCUBE AXIS Systems*, Pasadena, CA (December 1985).

40. K. S. Fu and T. Ichikawa, (eds.), "Special Computer Architectures for Pattern Processing," CRC Press, Boca Raton, FL (1982).

41. K. Preston and L. Uhr, "Multicomputers and Image Processing," Academic Press, New York (1982).

42. M. H. Raibert and J. E. Tanner, "Design and Implementation of a VLSI Tactile Sensing Computer," *Int. Jour. Rob. Res.*, FBI, #3, 3 (1982).

43. M. C. G. Hall, "The Human Analog Vision System," personal communication (December 1985).

44. W. J. Book, "Recursive Lagrangian Dynamics of Flexible Manipulator Arms," *Int. Jour. Rob. Res.*, **3**, #3, 87 (1984).

45. M. G. Forrest, S. M. Babcock et al., "Control of a Single Link, Two-Degree-Freedom Manipulator with Joint Compliance and Actuator Dynamics," *Proc. of 1985 Internat. Comput. in Eng. Conf*, **1**, 189-197 (August 1985).

APPENDIX A

AVOIDING REDUNDANCIES OF "TYPE A" (Redundancies related to Permutations of Processor Indices, and Redundancies related to Permutations of *spt* Indices)

ROSES specifies schedules using processor indices p and selection-point indices *spt*. As noted in Section 111.2, possibilities for redundant schedules arise because of the high degree of symmetry among processors p, and because there are no physical differences implied by differences in the *spt*-sequence of $[t,p,e]$ triplets identical in t.

What kinds of schedules, besides those completely identical in every detail, are considered redundant? To answer this, we first note that any ROSES triplet-series schedule may be specified as an unordered set of quartets $[spt,t,p.e]$. ROSES considers as redundant any two schedules made up of quartet sets differing only in p-labels and *spt*-labels. In other words, two schedules would be considered redundant if both involved the identical set of pairs $[t,e]$. Such redundancies – which we call "$[t,e]$-set redundancies" or "type-A redundancies" – are the kind discussed here. (For some other kinds, see Appendix B.)

Throughout our discussion of redundancies, we are concerned not so much with complete schedules, but rather with incomplete schedules – i.e., schedules under construction. We want to avoid constructing, and working to extend, incomplete schedules which offer the same opportunities for subsequent $[t,e]$ assignments as are offered by schedules already constructed.

To explain how ROSES avoids $[t,e]$-set redundancies, we start by examining some subclasses. In that connection we shall display a list of operations A1, A2, A3. (As will be shown following the list: In our descriptions of A2 and A3 the parts following the dashes need not be read carefully!) Two ROSES schedules would be $[t,e]$-set redundant with each other if and only if their series of *spt*-ordered triplets differed by any transformation combining operations of the following kinds A1, A2, A3:

OPERATION A1. For the indices p, any overall permutation.

OPERATION A2. For the indices p within an *spt*-ordered subseries of same-t triplets $[t,p,e]$, any permutation –
and then for the processor indices within any subsequent triplets, exactly the same permutation (so that each processor continues to get e-assignments at times appropriate to the pertinent time-lengths $L(e)$).

OPERATION A3. For the indices e within an *spt*-ordered subseries of same-t triplets $[t,p,e]$, any permutation –
and then for the indices p within any subsequent triplets, the permutation that, if applied to the original subseries of same-t triplets, would have produced exactly the same combination of $[p,e]$ pairs as were produced by the aforementioned permutation of indices e. (Here again, the permutation applied to "subsequent triplets" is to arrange that each processor continues to get e-assignments at times appropriate to the pertinent time-lengths $L(e)$.)

Concerning A1: The class of permutations being considered is covered by A2 with $t = 0$. Concerning A3: Within a subset of same-t triplets having a fixed series of *spt*-labels, permutation of the e-indices is equivalent to combining a permutation of the p-indices with a permutation of the *spt*-indices. For each of A2 and A3: If the permutations described preceding the dash were applied, then to take care of all subsequent permutations described after the dash, the ordinary ROSES constraint-satisfying procedures would suffice – within a transformation combining operations of type A1, A2, and A3.

Therefore, to avoid redundancies associated with A1, A2, A3 all that ROSES needs to do is: restrict every same-t subseries of triplets to a unique choice for the *spt*-sequence of p-indices involved, and to a unique choice for the *spt*-sequence of e-indices involved. Any unique pair of sequences will do, provided of course that the resulting series satisfies the input problem's constraints. (But ROSES will always satisfy those constraints.)

ROSES does choose and use such unique *spt*-sequences of p indices and e indices, disallowing others. In the subsections below, we sketch the procedures used to restrict these sequences. (Of course, care should be taken so that, in the process of eliminating redundant schedules, we do not inadvertently also eliminate some schedules which would be non-redundant with respect to those retained. This matter too will be addressed below.)

Redundancy-Avoiding p-Sequencing Procedure

The redundancy-avoiding p-sequencing procedure uses a "processor ready-time list": a list of processor indices arranged in nondecreasing order of ready time, where ready time is the time that each processor will complete its present assignment. The choice of a unique p-sequence within a series of same-t triplets, in the schedule, depends upon the choices made for sequencing p-indices within a series of same-ready-time members in the processor ready-time list.

The initial ready-time list, describing the state when all processors are idling and all have ready time $t = 0$, is arranged simply in order of p. Subsequently, whenever a processor p' is assigned a task or an idling period, the position of p' in the ready-time list is updated to reflect the fact that its ready time is changed to the forecasted completion time of its new assignment. Suppose that the new ready time of p' is equal to that associated with some sublist γ' of other processor-indices within the existing processor ready-time list. Then the position of p' is updated according to these rules: A processor p' that starts a zero-length task retains its position at the very beginning of the complete list – and so, retains its position at the very beginning of the sublist γ'. A processor p' that starts any other kind of assignment is inserted after the end of the sublist γ'.

At each *spt*, the very first processor in the ready-time list is assigned. By combining this rule with the above-described initial-position rule and position-updating rules, we find these rules for p-sequence within the schedule: If a processor starts a zero-length task at spt', then that same processor is reassigned at $(spt' + 1)$. Otherwise, processors which are ready at the same time are reassigned in the same order in which they were previously assigned. At $t = 0$, processors are assigned simply in order of p, with the following exception (which was ignored in our opening paragraph of III.6): If there are zero-length tasks ready at $t = 0$, and if processor p' is assigned at spt' to start a zero-length task at $t = 0$, then that same processor p' will be

reassigned at (spt' + I) and processor (p' + I) will not be assigned until the next selection point after processor p' is given a nonzero-length task.

However, the details of these results for p-sequence in the schedule are not important. In fact, the order of processors, within a same-t sublist of the ready-time list, not only has no effect upon the set of [t,e] pairs in a schedule; it has no effect on the spt-order of these pairs, either. The important point is that the above-described p-sequencing procedure – which happens to involve a ready-time list – does determine a unique p-sequence in the schedule's subseries of same-t triplets.

Further description concerning the processor ready-time list is given in section III.7.2, which discusses the data structures PR and R. Details of the redundancy-avoiding p-sequencing procedure are given sections 1V.5.3 and lV.5.4, which present outlines of subroutines CheckE and ForwPR.

Of course, to complete the job of avoiding $[t, e]$-set redundancies, a unique choice must be made also for the e-sequencing within each subseries of same-t triplets in the schedule. This e-sequencing choice is discussed next.

Independence of e-Sequencing and p-Sequencing Procedures

The choice made for the spt-sequence of p-indices, within a schedule's same-t subseries of triplets, does not influence the choice made for the spt-sequence of e-indices. The reason is connected with the following feature of ROSES: Before making any assignments at a given spt', ROSES determines the entire set of tasks known to be ready at spt'. Here a task ready at spt' means a task whose precedence requirements are fulfilled by predecessor tasks each of which (i) was assigned in a triplet with tag spt less than spt', and (ii) had or has a ready time $\leq t_{(spt')}$. Note that there is no requirement for the predecessor tasks to have finished at $spt < spt'$ – i.e., no requirement for the predecessor tasks to have been in triplets with processors that were given other e-assignments at $spt < spt'$.

Redundancy-Avoiding e-Sequencing Rule

Consider the group of tasks e_i ready at some single point spt' – each of them assigned at its own spt_i such that $spt_i \leq spt'$ although every t_i has the same value, $t_i = t_{(spt')} \equiv t'$. ROSES restricts its schedule-construction to schedules in which these e-assignments are spt-sequenced in order of decreasing "favor" – where favor is defined in Section III.6 beginning near Eq. (4).

As in the case of the p-sequencing results, the particular unique choice made, for spt-sequence of e-indices, is unimportant in the sense that "any unique pair of p-sequence and e-sequence will do" for the purpose of avoiding $[t, e]$-set redundancies. However, the e-sequencing rule stated above has three computational conveniences worth mentioning. First, it has the overall convenience of using exactly the same e-ordering convention as ROSES uses to heuristically determine the sequence in which to consider truly different schedules. Second, the redundancy-avoiding e-sequencing rule has this special feature: all tasks starting at t are assigned prior-in-spt to all idling periods starting at t. This special feature facilitates the assignment of idling periods, because it guarantees that when ROSES assigns idling periods, ROSES will have all the information it needs to determine those idling periods' completion times. (As noted in III.3, these completion times are set equal to the earliest time

after t that any task-assigned-at-or-before-t will finish.) Third, because "favor" among tasks is f, L favor as defined near Eq. (4), the redundancy-avoiding e-ordering convention has a special convenience associated with zero-length tasks. This special convenience is explained just below.

Suppose that among the above-mentioned tasks e_i ready at *spt* there is at least one zero-length task; call it e_o. Suppose that this e_o is assigned at t', and of course then finishes at t', and by so finishing renders at least one more task ready at t'. Suppose, though, that the assignment of e_o is not made at the smallest *spt* for t'. In that case: at the program-execution-point where ROSES reaches the smallest *spt* for t', ROSES will have too little information to "know" the entire set of tasks-ready-at-t'. Without knowing the entire set of tasks ready at t', ROSES cannot find their order of decreasing f, L favor. Instead, ROSES must reach the later *spt* that is associated with adding the triplet incorporating e_o, before knowing which if any additional tasks are to be included in the set of ready-at-t' tasks. Despite this delay in knowledge, there is little complication and no need to change the redundancy-avoiding e-ordering rule given above for *spt*-sequencing e-assignments starting simultaneously at t'. This is explained as follows. Because of the way that f, L favor is defined, any task whose precedence requirements are newly satisfied by the completion of a zero-length task necessarily has f, L favor below that of the zero-length task itself. Therefore the delay in knowledge is never so great that ROSES has assigned a less-favored task at a smaller *spt* than it should have. Consequently, the stated conventional sequence, decreasing order of favor, is easily effected without undoing any assignments. The only complication is that ROSES must add the tasks-made-ready-by-e_o to the set of other ready tasks lower in favor than e_o.

The above considerations are, however, not quite the end of the story concerning zero-length tasks and redundancies. The existence of zero-length tasks warrants ROSES procedures that eliminate another kind of redundancy, not covered in this Appendix A. Discussion of that other kind is included in Appendix B, under B2.

The implementation of the redundancy-avoiding e-ordering rule is discussed in Section III.7, in connection with the data structures EA and A. Details are given in the outlines of subroutines ForwEA and CREATE in Sections IV.5.5 and IV.5.6.

Final Note: Cases Where Redundant Schedules Can Be Useful

For some applications, it may be appropriate to turn off or circumvent some of the ROSES code's redundancy-avoiding procedures. That could be useful in situations where the following conditions (i) and (ii) hold:

i. There are symmetries in the mathematical problem handled by ROSES (e.g., perfect symmetry among processors) which do not exist in the real-world problem; thus, ROSES neglects certain asymmetries in the real-world problem. However,

ii. The ROSES-neglected asymmetries in the real-world problem can be taken into account, after the ROSES calculation, by altering or reinterpreting the ROSES-produced schedules so as to discriminate between schedules considered "essentially redundant" by ROSES.

APPENDIX B

REDUNDANCIES OF "TYPE B"
Redundancies Related to Properties of Tasks)

As in Appendix A, we are concerned here with redundancies of partially completed schedules, i.e., schedules under construction.

General Differences Between Type-A Redundancies and Type-B Redundancies

Here in Appendix B we discuss how ROSES avoids redundancies associated with task-properties (*e*-properties). In contrast, in Appendix A we were concerned with redundancies associated with the *p*-properties and *spt*-properties. This difference leads to the following difference: Within a set of type-A-redundant schedules, all schedules have exactly the same set of $[t, e]$ pairs. Within a set of type-B-redundant schedules, the schedules have different $[t, e]$ pairs but have other similarities that imply redundance of opportunities for total TIME_to_GOAL.

Unlike Appendix A, this Appendix B uses some search-tree terminology (explained in III.4) and task-graph terminology (explained in III.5).

Type-A redundancies are avoided by directly limiting the search tree. Type-B redundancies are avoided in other ways. Some are avoided by traversing a branch and then rejecting the resulting schedule; others are avoided by avoiding the traversal of existing search-tree branches.

There are two subtypes to be considered here: B1 and B2.

Redundancies of Type B1

Suppose that two tasks are of equal length, share the same head, and share the same tail. Then substituting one of these tasks for the other, in a schedule, will make no difference in timing or readiness, and so will had no effect on the opportunities for performance of other tasks at subsequent times t in the schedule. In fact, exactly the same comments hold true if the two tasks merely have the same length and head, even if they have different tails. Therefore, ROSES classifies as "not worth continuing" any schedule which differs from an already-constructed schedule only by the kind of substitution just described.

In this B1 case, the redundant schedule-paths exist in the search tree. For B1, eliminating a redundant schedule falls in the category of traversing a branch and then deciding not to continue the redundant schedule. The ROSES-code implementation is described in Section IV.5.3, in our outline of subroutine CheckE, steps 3 and 6.

Redundancies of Type B2

Consider two schedules, ALPHA and BETA, which involve the same combination of $[t, e]$ pairs except for differences in pairs involving zero-length tasks. Suppose that schedule ALPHA has assigned all of the zero-length tasks that schedule BETA has assigned, and suppose further that in each case of a zero-length task e_o assigned by

both ALPHA and BETA, schedule ALPHA has assigned that task e_o at least as early as schedule BETA has assigned it. Under those stated conditions: At the time-point of its latest assignment, schedule ALPHA offers all of (and perhaps more than) the opportunities offered by schedule BETA at that same time-point. That is, schedule BETA may be more restrictive in the opportunities it offers, because a delay in assigning a zero-length task e_o will delay the opportunity of assigning nonzero-length successor tasks e' for which e_o would satisfy precedence constraints. However, schedule ALPHA offers all of (and in some cases more than) the opportunities that schedule BETA offers, because even if a zero-length task is assigned early, its successor tasks e' can still be delayed. Therefore to avoid redundant schedules, ROSES should not construct both schedule ALPHA and schedule BETA.

To eliminate redundancies of the kind described just above, ROSES restricts its considerations to schedules in which every zero-length task is assigned at the earliest time t consistent with:

- the $[i, e]$ pairs for nonzero-length-e assignments,
- the e-sequence rule of Appendix A,
- the precedence constraints,
- the processor-number constraint, and
- the formal "non-interruption" constraint.

For the present discussion, the essence of the last-mentioned constraint may be written this way: A processor assigned to a nonzero-length e at spt_i cannot be assigned to any other e (even a zero-length e) at any $spt > spt_i$ having $t_{(spt)} = spt_i$.

To implement the entire "earliest-t ..." positioning rule described under B2: When ROSES is backtracking and comes to a vertex at which the last-selected branch was a zero-length task, ROSES backtracks yet further instead of considering the substitution of a lesser-favored branch. See step 3.4 in our Section IV.5.1 outline of Main.

Final Note: Cases Where Redundant Schedules Can Be Useful

The "Final Note" written at the end of Appendix A is applicable to these Appendix B redundancies also.

APPENDIX C

MORE DETAILS ABOUT TASK-BRANCH SETS AND THE *e*-ALTERNATIVES STRUCTURE

The task-branches stemming from a vertex correspond to all the tasks e that are "ready at vertex *spt*," minus some which are excluded in order to acid the construction of redundant schedules. This appendix continues Section III.6.3 by discussing:

i. the relation between the set of task-branches stemming from vertex (spt – 1) and the set stemming from vertex *spt*, and

ii. some details about the arrangement of these task-branches' e-indices within the long list A of the e-alternatives structure EA.

This appendix's statements concerning (i) are justified by ROSES properties described in text-sections up through III.7.3, and Appendices A and B. The statements concerning both (i) and (ii) are consistent with the ROSES-code details described in Section IV.

If $t_{(spt)}$ differs from $t_{(spt-1)}$, then the task choices at *spt* consist of all the tasks that have been made ready at $t_{(spt)}$ because of precedence requirements fulfilled by tasks started at vertices $spt' < spt$. If $t_{(spt)}$ is the same as $t_{(spt-1)}$, then the aforementioned relation holds except that, because of redundancy-avoiding considerations, tasks of favor greater than $e_{(spt-1)}$ are disallowed at vertex *spt*. As noted in Section III.7.3, a task-list is considered empty if $FIRSTia(spt)$ exceeds $LASTiaPLUSone(spt) - 1$. These general statements lead to the following special statements about the relations between branches at neighboring vertices.

If $t_{(spt)}$ differs from $t_{(spt-1)}$, and if $t_{(spt-1)}$ is unequal to the t at any other vertex of the schedule, then the task-choices at vertex *spt* consist of: all the task-choices at vertex (spt – 1) minus any task-choice $e_{(spt-1)}$ started at vertex (spt – 1), plus any tasks whose precedence requirements are newly completed at $t_{(spt)}$ by tasks assigned before *spt*. The task-list

$$A(FIRSTia(spt - 1)),...,A(LASTiaPLUSone(spt - 1) - 1)$$

for vertex (spt – 1), and the task-list

$$A(FIRSTia(spt)),...,A(LASTiaPLUSone(spt) - 1)$$

for vertex *spt*, are contiguous within A (if non-empty) and are physically distinct from each other (if non-empty). That is, the location of $A(FIRSTia(spt))$ coincides with the location for $A(LASTiaPLUSone(spt - 1))$. Either or both task-lists may be empty.

If $t_{(spt)}$ is equal to $t_{(spt-1)}$, then the relations of task-branch sets are as described just in the preceding paragraph, with these two exceptions:

i. Tasks more favored than $e_{(spt-1)}$ are disallowed as branches from vertex *spt*, because of redundancy-avoiding considerations; and

ii. If $e_{(spt-1)}$ was anything but a zero-length task, then the task-list
$A(FIRSTia(spt), ..., A(LASTiaPLUSone(spt) - 1)$

for vertex *spt* is not physically separate from the task-list
$A(FIRSTia(spt - 1), ..., A(LASTiaPLUSone(spt - 1) - 1))$

for vertex $(spt - 1)$. Instead, the task-list for vertex *spt* coincides with the upper part of the task-list for vertex $(spt - 1)$, and the location of $A(LASTiaPLUSone(spt))$ is the same as the location of $A(LASTiaPLUSone(spt - 1))$. Furthermore, in this case where *e* is other than a zero-length task, only the following emptiness conditions can occur: Both task-lists may be non-empty, or both may be empty, or only the *spt* task-list may be empty.

If $t_{(spt)}$ differs from $t_{(spt-1)}$ but $t_{(spt-1)}$ is the same as the t for some preceding vertices in the schedule-path, then the *e*-choices at vertex *spt* consist of: the *e*choices at the lowest-indexed vertex *spt* having $t_{(spt')} = t_{(spt-1)}$, minus any tasks assigned at or after *spt'* but before *spt*, plus any tasks made newly ready at *spt* by virtue of the completion of their precedence requirements at $t_{(spt)}$ by tasks which were assigned before *spt* and which finish at $t_{(spt)}$. The task-list
$A(FIRSTia(spt - 1)), ..., A(LASTiaPLUSone(spt - 1) - 1)$

for vertex $(spt - 1)$, and the task-list
$A(FIRSTia(spt)), ..., A(LASTiaPLUSone(spt) - 1)$

for vertex *spt*, are contiguous within *A* (if non-empty) and are physically distinct from each other (if non-empty). That is, the location of $A(FIRSTia(spt))$ coincides with the location for $A(LASTiaPLUSone(spt - 1))$. Either or both task-lists may be empty.

APPENDIX D

RELATION OF THIS REPORT'S TERMINOLOGY TO ROSES-CODE TERMINOLOGY

This report's terminology was chosen to differ from the ROSES-code terminology, in cases where such differences seemed to allow simpler clearer explanation of the essential ideas behind ROSES data structures and algorithms.

Table D1 relates the two terminologies. For brevity, Table DI and this Appendix's text use some algebraic notation in place of pure computer-code notation. For example, the table lists pproc($3p'+1$) rather than pproc (3*$pprime$+1).

The coded arrays $G(...)$ and $pproc(...)$ deserve further discussion, which appears in the text below.

The Array $G(...)$

The coded array $G(...)$ may be viewed in terms of its quartets

$$G(4e'-3),\ G(4e'-2),\ G(4e'-1),\ G(4e')$$

where

e' represents a task-index;

$G(4e'-3) = tail(e')$;

$G(4e'-2)$ is a "pointer"; it stores that index I for which

$$G(I) = tail(previous\_e\_for\_same\_head(e'))$$

$G(4e' - 1) = head(e')$

$G(4e')$ is a pointer; it stores that index J for which

$$G(J) = head(previous\_e\_for\_same\_tail(e'))$$

Another way of stating the relation is:

$G(4e'-3)$ = tail(e');

$G(4e'-2)$ = 4 ($previous\_e\_for\_same\_head(e')$) - 3;

$G(4e'-1)$ = head(e');

$G(4e')$ = 4 ($previous\_e\_for\_same\_tail(e')$) - 1.

Table D1. Relation of this report's terminology to ROSES-code terminology

Variable-name in this report[a,b]	Equivalent in the ROSES Code
A(ia)	*Estk*(...)
$CURRENTia$(spt)	*ptstk*(spt)
$DMAX$	*MaxDelay*
e	*(see footnote c)*
efinishing(spt)	*Fstk*(spt)
esched(spt)	*esched*(spt)
$evalue(p')$	$pproc(3p'+1)$
$f(e')$	$fedge(e')$
$FIRSTia$(spt)	*botsk*(spt)
goalnode	*goalnode*
$head(e)$	$G(4e\text{-}1)$
$iaCURRENT$	*stkpt*
$iaFIRST$	*stkbot*
$iaLASTplusONE$	*stktop*
$LASTiaPLUSone$(spt)	*topstk*(spt)
N_e, NE	*nedges-1*
N_p, NP	*nprocessor*
$nfie(u)$	$na(u)$
$nfieus(v)$	$ma(v)$
$nfieuts(v)$	$maa(v)$
$nfoe(v)$	$nb(v)$
p	*(see footnote c)*
$pbak(p')$	$[pproc(3p'+3)\text{-}1]/3$
$pfwd(p')$	$[pproc(3p'+2)\text{-}1]/3$
$previous\_e\_for\_same\_head(e)$	$[G(4e\text{-}2)+3]/4$
$previous\_e\_for\_same\_tail(e)$	$[G(4e)+1]/4$
psched(spt)	*psched*(spt)
sold	*sold*
spt	*spt*
t	(see footnote c)
$tail(e)$	G(4e-3)
$TIME\_to\_GOAL$	*tsolution*
tneztfcp	*tnext*
$tNEXTinSCHED$	*tnew*
told	*told*
$tready(p')$	$tproc(p'+1)$
trequired	*trequired*
u	u
v	v
Routine in this report	**Routine-name in the ROSES code**
CheckE	*ChooseEdge*
ForwEA	*UPSTAT*

[a] This column concentrates on cases for which this report's terminology differs from the code's. (Included, however, are a few of the cases for which this report's terminology coincides with the code's.)

[b] Names with suffixes BOP, INT, and EOP are omitted because ROSES-code counterparts to them are the same as if these suffixes were absent. (In short: ROSES-code variables have dynamically changing meanings, encompassing meanings which this report distinguishes among by attaching suffices BOP, INT, EOP.)

[c] More details about t, p, e are given in Sec. IV.3 below its last boldface subheading.

The Array pproc(...)

The coded array $pproc(...)$ may be viewed in terms of its triads

$pproc(3p'+1), pproc(3p'+2), pproc(3p'+3)$

where

p' represents a processor-index;

$pproc(3p'+1) = evalue(p')$;

$pproc(3p'+2)$ is a pointer; it stores that index I for which

$pproc(\mathrm{I}) = evalue(pfwd(p'))$;

$pproc(3p'+3)$ is a pointer; it stores that index J for which

$pproc(\mathrm{J}) = evalue(pbak(p'))$;

Another way of stating the relation is:

$pproc(3p'+1) = evalue(p')$;

$pproc(3p'+2) = 3pfwd(p') + 1$;

$pproc(3p'+3) = 3pbak(p') + 1$.