A Task and Resource Scheduling System for Automated Planning

David P. Miller

TR 87-1

A Task and Resource Scheduling System for Automated Planning

David P. Miller

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061
Phone (703) 961-5605
Net: miller%vtcs1@bitnet-relay.arpa

Abstract

Planning is done at both the strategic and tactical levels. This paper classifies some previous planning techniques into these different levels, and details some of their problems. A planning technique known as heuristic task scheduling is then presented along with a planner arhitecture that integrates task-scheduling with more traditional techniques to form a system that bridges the strategic/tactical division.

1 Introduction

Most planning work of the past has concentrated on formal systems with limited practical abilities. While these systems have laid the groundwork for planning research, they have dealt with idealized, static domains that do not well represent a modern factory or other real-world situation. Therefore, work on artificial intelligence planning has had little practical influence on task planning, scheduling, or other issues of interest to operations research. Recently, it has become clear that AI search techniques can be used to attack some OR problems [8]. The research and system described in this paper use AI search techniques as a bridge to link the (mostly) theoretical work on artificial intelligence planning with practical problems found in arranging, scheduling and allocating resources for real-world factory and robot applications.

The remainder of this section provides a brief historical perspective and classification of some of the previous work in planning. The need for search in planning will be demonstrated, as well as the need for a new type of representation. The next section provides a description for a new planning module: the Heuristic Task Scheduler. Some of the techniques and algorithms used in the HTS are described. The third section describes how the HTS can be integrated into a more general type of planning system. An implemented system is described along with some ideas for future work.

1.1 Strategic and Tactical Planning

Robot planning may be thought about as two distinct activities:

- making strategic planning decisions
- choosing the proper tactics, based on the actual situation, for executing those strategies.

Hierarchical least-commitment planning systems (e.g., Noah [19] and Nonlin [21]), which have dominated planning research for the past decade, concentrate exclusively on the strategic part of planning. Part of the reason for this lopsided research is that tactical planning decisions are only important in dynamic domains. The highly rarefied domains that most planners operate in (such as the blocksworld) leave little room for tactical decisions to be made.

For example, in the blocksworld domain there is really only one way to go about accomplishing a goal such as **achieve block A on block B**. There may be several different "plans" for accomplishing this goal but all of those plans contain the steps:

1. **clear block A**
2. **clear block B**
3. **stack A on B**

The tactical details (e.g., exactly how to clear the blocks) are left to the robot's imagination. Leaving the actual execution details out of a plan is ok in the blocksworld but may leave the majority of the planning work undone for more realistic domains.

It may at first appear that the tactical details of a plan could be fleshed out by simply recalling the planning system until a sufficient level of specification had been reached. Unfortunately, most planning systems have been based on two assumptions that do not hold once a certain level of planning detail has been reached. These assumptions are:

1. plans involving conjunctive goals can have the subplans for each goal expanded independently of one another
2. information needed to perform plan expansions down to the level of primitive actions is available at plan time.

A simple example should help to show the lack of reality in the two assumptions above. Imagine a simple factory that produces a single plastic product that is made of two subassemblies. The top-level strategy for creating the product is to accomplish three steps:

- **create subassembly A**
- **create subassembly B**
- **join the subassemblies while both are still warm**

Assume that the factory contains one furnace and one plastic mixer, and that either subassembly can be created by melting a plastic blank into a mold or by mixing plastic liquid and a catalyst. If the melting plan is chosen for subassembly A then the mixing plan **must** be chosen for subassembly B. If it is not, then one of the subassemblies will be cool by the time the other is ready. Therefore, the first assumption, that subplans may be refined independently from one another is clearly not always valid.

If the expansion of the melting plan is:

1. **put blank in mold**
2. **put mold in furnace**
3. **turn furnace on**
4. **turn furnace off as soon as plastic bubbles**
5. **remove mold**

then the second assumption is also not valid. A plan, such as that immediately above, cannot be reduced to primitive actions before the plan is well underway. When should the actor look at the melting plastic? When should the furnace be turned off. These questions cannot be answered until the actions are actually in progress. A plan to handle this situation must contain calls to gather sensory information. It may need loops and conditionals to detail the necessary behaviour. As we shall see in the remainder of this section, such structures do not fit well with current planning strategies. Therefore, a new planning paradigm, simulation-based planning, is needed.

1.2 Situation-Based Planners

Situation-based planners such as Strips [6], and its many decendents, try to derive a sequence of actions that will produce a state of the world where the goal of the task is achieved. Situation-based planners represent the state of the world as a conjunction of predicate-calculus assertions. The goal-state is represented similarly. The planner compares the representations of the two states, notes the differences, and applies a sequence of *operators* to the initial-state in order to transform it into the goal-state.

In order for the planner to know which operator to apply, it maintains lists, associated with the operator, that specify the predicate-calculus assertions that should be added and/or deleted as each operator is put into the plan. Each operator can also have a list of assertions that must be in the state of the world before it can successfully be applied. For example, the (stack A B) operator may require that (clear A) and (clear B) must already exist in the world state before stack may be applied. These preconditions are added to the plan before the operator that actually is chosen to reduce a difference between the current world-state and the goal-state.

The features that make situation-based planners practical to implement unfortunately also make them unsuitable for most real-world problems. Primary among these features is the dependence on first-order predicate-calculus as the representation scheme. The predicate-calculus is necessary to make difference comparison and operator selection practical because it makes such a comparison well-defined and reduces the actual comparison operation to simple patern matching. Unfortunately, such a representation scheme is not readily suited for handling temporal factors, quantitative issues, or uncertainty [3]. Extensions to predicate calculus have been suggested by Allen [2], and McDermott [14] to allow temporal, resource and default reasoning. However, further work has shown that these extensions can provide the basis for a system of limited abilities at best [9].

Another difficulty with the situation-based planners is that they are neither strategically or tactically oriented. The lack of high-level strategic guidance in forming a plan can greatly increase the amount of search that is necessary in selecting the proper operators to achieve a goal. If several goals are to be achieved in conjunction, and not any order of solution will suffice, then it is only luck that will guide a Strips-like system to an efficient solution rather than trying to reduce all the differences independently and probably most inefficiently. The Abstrips system [20] was a first attempt at adding hierarchical organization to the search strategy of a situation-based planner.

While situation-based planners deal almost exclusively with low-level operators, they are not necessarily practical tactical planners. Their method of reducing differences does not at all guarantee that the reduction of one difference will not add other differences.

1.3 Hierarchical Least-Commitment Planners

Hierarchical least-commitment planners were developed to overcome some of the shortcomings of situation-based planners. Chief among these shortcomings were the situation-based planners' inability to deal with any sort of interaction between conjunctive subgoals (the first planning assumption). Hierarchical planners such as Noah and Nonlin form plans through the successive refinement of an initial task statement. The task is indexed into a library of plans, and an appropriate plan is selected. This plan consists of a partial order of subtasks to be achieved, the conjunctive solution of which should achieve the original task. Each of these *subtasks* is then indexed into the plan library and similarly expanded. Some sort of table of predicted side-effects is maintained as each plan is retrieved from the library.

As the plans are expanded, usually in a breadth-first manner, the side-effects are factored into the required preconditions for each plan at that level of expansion. Should a side-effect contradict a precondition for some other plan, then the planner will attempt some sort of fix to resolve the problem. This fix usually takes the form of putting in an explicit ordering link between the two plans in conflict forcing the side-effect to occur after the precondition is met. This allows hierarchical planners to solve some problems considered unsolvable by the situation-based planners. However, the case where a side effect contradicts a precondition at the same level of plan expansion is but one of large set of conditions where goals interact [24].

To avoid some of the problems from other types of goal interaction, if no conflict is detected at that level of expansion and there is no ordering of tasks explicit in that part of the plan, the subtasks are left unordered. This has the advantage of allowing more flexibility in subsequent levels of expansion. Unfortunately, as discussed previously, leaving plan steps unordered makes it impossible to detect certain types of context-dependent side-effects that can cause problems later on.

While hiearchical planners can solve many problems unsolvable by situation-based planners, situation-based planners have an important advantage over hierarchical least-commitment planners. That advantage is that if the representation used by the situation-based planner is sufficient to capture the interaction between the plan steps, if the situation-based planner is able to derive a plan, and if the information that the planner used was correct, then the plan is more likely to be feasible then a similar plan produced by a hierarchical system. The reason being that the situation-based planner has performed, through the addition and deletion of the relevant predicate-calulus assertions, a simple linear simulation of the plan. And any simulation is more likely to discover context-dependent interactions then is possible through debugging a partially-ordered plan hierarchy.

2 Simulation-Based Planning

The planning strategy used in this research is to search through a variety of simulations of the execution of different plans, and select the plan with the most successful simulation. By simulating the execution of a plan the planner can take a more global perspective in making planning decisions than is possible in a traditional hierarchical planner. It is impossible to *ignore* deleterious interactions among subplans when the plan is being generated and simulated in a totally-ordered manner, rather than in a partially-ordered hierarchical manner.

A detailed simulation allows the limits of reliability to be accurately calculated. As each action in the plan is simulated, anticipated errors in movement, timing, and resource consumption can be propagated. The points in the plan where errors accumulate above acceptable levels are the places in the plan where feedback will be needed and some real-time tactical decisions might be necessary.

The planner's job is to create a sequence of primitive actions that, when executed by the robot, will accomplish the desired task. The plan thus produced must

- Be free of adverse interactions among the plan steps
- Have all the necessary resources explicitly assigned to each task
- Have the plan steps temporally coordinated with one another
- Explicitly include state-transition tasks (e.g., the tasks necessary to bring any resources required for a particular task into the state that they are required for that task.)

Planning, by this definition, bears some resemblance to scheduling, and may be accomplished by taking a scheduling approach called *task scheduling*. Task scheduling is accomplished through the incremental extension of selected schedule *prefixes* until a complete schedule is found. A schedule prefix of length n is a partial schedule that starts with the first task to be done, and continues sequentially for $n - 1$ additional tasks. A heuristic task scheduling program (HTS) has been implemented which performs task scheduling over a partially-ordered set of tasks. The HTS effectively combines features from both the situation-based planners and the least-commitment planners. The idea of a totally-ordered search is a major part of the situation-based planning approach. The HTS augments this approach with the idea of searching through plan-space, rather than the operator-space that Strips searched through. By searching through plan space, and by actually performing a more detailed simulation, the HTS has much more information available to it to guide its search than is available to a system such as Strips.

Most scheduling problems are NP-complete in their computational complexity. The task scheduling problem is no exception. In order to create a feasible planning system, it is therefore necessary to attack the task scheduling problem from a heuristic approach. The next section presents details of the heuristc algorithm used by the HTS.

2.1 Searching for a Good Schedule

The role of the HTS is to convert a partial ordering of tasks into an efficient executable total ordering. The total ordering has associated with it a sequence of temporal intervals; one interval associated with each step in the ordering. This allows the robot to monitor its progress through the plan as well as allowing the total ordering to be integrated with other schedules that are to precede and follow it. In order to ensure that the ordering produced by the scheduler is a good one, the scheduler performs a limited simulation on the ordering. This system simulates tasks and calculates the state that would result from the execution of those tasks. Additionally, this system uses the results from partial simulations to efficiently order tasks that are to be performed later in the schedule. Towards this end the scheduler searches through the space of possible task orderings in an attempt to find a reasonably efficient ordering. Since the number of possible schedules rises exponentially with the number of tasks being scheduled, the system relies on scheduling heuristics [22], [18] to trim the search tree to a reasonable size. The search heuristics rely on the following properties that are associated with each task:

- The location and type of resources required for the task
- Partial ordering constraints with respect to other tasks
- The absolute execution window for the task (e.g., do this between 9:00 and 11:00am)
- Metric ordering information with respect to other tasks (e.g., do B between fifteen and twenty minutes after completing A)

These properties of the tasks, and their relationships with one another, can be used to reduce the total scheduling space in a totally domain-independent way. The reduced search space can then be searched using further domain-independent and dependent heuristics to help guide the search.

A detailed description of the basic scheduling process is presented next. Immediately following the scheduling algorithm are descriptions of the data structures manipulated by that algorithm. Finally, the methods in which prospective schedules are rated and chosen, are discussed.

2.2 The Scheduling Process

The HTS uses a heuristically guided best-first search for creating the plan schedule. For this type of search, all the conceivable schedules can be thought of as being arranged in the form of a tree as in Figure 1. Of course to make the schedule search efficient, as little of the search tree as possible is generated.

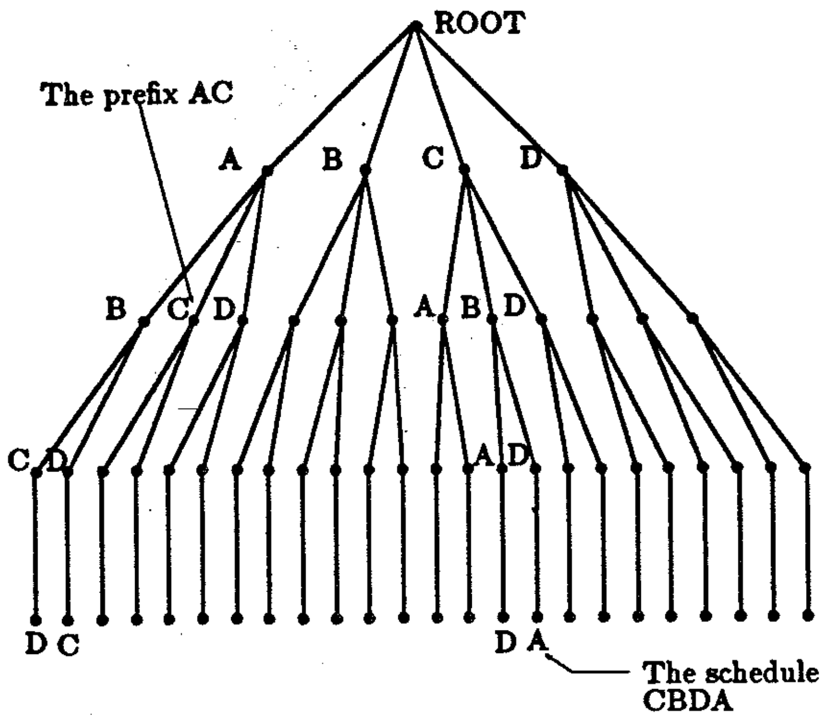


Figure 1: A Scheduling Tree for Four Tasks: A,B,C, & D

There are two basic strategies for limiting the search space:

- generating only the feasible schedules
- generating only the promising schedules.

The HTS uses a combination of both of these strategies.

Scheduling is done, in this system, by incrementally extending selected schedule *prefixes* until a complete schedule has been found. At each iteration through the search process the system selects the prefix that looks most promising (see section 2.7). The feasible extensions to that prefix are then generated and the process repeats. If there are no legal extensions to the prefix then that prefix is deleted from the search space. Occasionally, during the generation of the prefix extensions a flaw will be found that may be common to a whole class of prefixes. When this is the case a critic will be generated (see section 2.7.3) and passed back to the search routines instead of the prefix extensions. The critic is then run over all of the prefixes under consideration. Those that share the flaw are detected and removed from the search space.

2.2.1 The Searcher

The HTS uses a heuristically guided state-oriented searcher for exploring the space of totally ordered plans. This type of search strategy involves generating partial schedules and saving the state produced by the partial schedule. The partially generated schedules are then rated, and the most promising one is expanded further. The prospects are then rated again, and the most promising schedule prefix is extended. Several different prefixes, in different states of completion, may be being examined at any one time. The quality of the search strategy is judged not only on the schedule it finally selects, but also on the number of schedule expansions it does. To reduce the number of schedule expansions the HTS search strategy is a combination of several search techniques such as: beam search, best-first, and branch and bound. How each of these techniques fits in the overall search strategy is described below.

One common situation which could cause a search-based scheduler to perform badly is when it is given a lightly constrained set of tasks where one task is very expensive to perform. In such a situation the scheduler might expand a prefix until it reached the point where it had to schedule the expensive task. This might sufficiently lower the prefix's viability score so that some less developed prefix looks more promising. The new prefix would then be expanded until it also had to add the expensive task. In the worst case this would continue until almost the full search tree had expanded and the expensive task was the only expansion possibility left for all of the possible prefixes. At this point the system would take the best scoring prefix and expand it with the expensive task, and then continue on. The prefix it finally chooses to make the expansion was probably the prefix it had been initially working with; all the subsequent work was just an exponential sink hole.

To overcome this problem the HTS only has a small fraction of the full search space available to it for backing up and improving the score. The limited search space causes the scheduler to follow its original "intuition;" the system cannot vacillate between possible prefixes because it does not have access to the list of alternatives.

The search space is constrained by putting limits on the *bushiness* of the search tree and the number of prefixes that are considered at each iteration. The bushiness, or branching factor, of the tree can be thought of as how many possible extensions there are to a given prefix. In a totally unconstrained scheduling problem there are $N - k$ possible extensions to each prefix where there are a total of N tasks to be scheduled, and the prefix in question is of length k.

The number of prefixes being considered by the search mechanism is the same as the number of leaves in the partially constructed search tree. In a poorly constrained problem of size N the number of leaves approaches $N!$. This performance can be drastically improved by using some sort of *beam search* technique as in [12]. *Beam search* uses an arbitrary cutoff of say ten, so that only the ten most promising prefixes are considered for extension at any time.

Unfortunately, putting limits like these on the search can cause the system to report that no feasible schedule could be found where in fact one might really exist. Limiting the overall number of prefixes being examined is especially bad in this respect, because in large problems the early level prefixes have large numbers of possible extensions, most with about the same score. In this situation, most or all of the limited number of prefixes being considered will have many early ordering decisions in common. If those decisions happen to have been wrong then the search will fail because the system had discarded the workable alternatives early on as not promising enough.

The best strategy appears to be a blend of restrictions on the bushiness of any branch in the search tree along with limits on the total search *front* — the set of prefixes that can be expanded upon. This gives the effect of performing several widely spaced beam searches each having an extra narrow beam. This strategy is preferable to a straight beam search because in task scheduling, if a prefix turns out to be unworkable it is quite likely that a significantly different prefix should be pursued. Using a normal beam search would require a very large beam width to ensure that a satisfactory answer was eventually found. The HTS's search uses the limitations on bushiness and the length of the search front, along with a new system for handling worst-case backtracking. To get the best results, the size of the front must be coordinated with the allowed bushiness of the search space. As the search progresses under a normal tree search algorithm, the number of possible prefixes rises, but at the same time the bushiness decreases. However, the two are not well balanced so that the number of nodes that could be considered peaks at level $N-1$. What is needed is an algorithm that keeps the number of nodes at a smaller and more constant level, yet still keeping a wide variety of possible solutions available to the system. This can be achieved by limiting the bushiness of the tree at any level k to an amount considerably smaller than than the total number of nodes being examined. A good heuristic appears to be to set an overall limit of $max(4, \log N^2)$ prefixes to be available to be examined and a bushiness of $max(2, \log N)$. On small problems ($N < 100$) there is sometimes not enough information available to the heuristic rating functions to allow logarithmic trimming of the search tree. So on small problems the bushiness of the tree is kept at two and the number of nodes under examination at any time is limited to four.

The prefixes and prefix expansions that are removed from the front by these limitations are place onto a *backup queue*. There they can be accessed by the system should it prove that no feasible schedule can be extracted from the possibilities available to it from the search front. The size of the backup queue can be varied to control the search time verses assurance of finding an answer tradeoff. For the experiments performed so far with the system, a backup queue size of $max(4, \log N^2)$ seems sufficient to ensure the construction of a reasonable schedule.

2.2.2 The Schedule Dependency Array

In order to perform a heuristic search there must first exist some coherent method for selectively generating possible extensions to the search tree. It is the SDA's job to produce, upon request, all of the *feasible* daughter nodes to a particular node in the search tree. A feasible node is one that produces a schedule prefix free of domain dependent and independent problems, i.e., one that meets all of the restrictions imposed on the schedule so far. In the event that there is more than one daughter node for the prefix that the SDA is expanding, it falls upon the SDA to perform an initial evaluation upon the possible expansions and to pass on only those that appear promising.

An SDA is a data structure designed to keep track of a particular schedule prefix's task interdependencies. The SDA consist of a dependency matrix and a set of status vectors which keep track of various properties about each task that has yet to be scheduled. With a minimal amount of computation on an SDA, the answers to the following questions can be derived about each task:

- Does the task being considered still need to be placed on the schedule?
- Is the current simulation time a legal time for a particular task to be scheduled?
- Have all of the prerequisites for the task in question been placed on the schedule?
- Are other tasks being executed in this state with which this task is supposed to be disjunctive?
- Can all of the spatial and resource restrictions be satisfied?

The SDA also indicates whether the particular prefix it is working on leads irrevocably into some sort of dead end. A dead end would be where some deadline violation or resource restriction must occur no matter how the prefix is expanded. The SDA serves as an organizational device from which information on the many possible relational objects can be quickly and efficiently derived.

The SDA can be thought of as a $N \times N$ matrix where N is the number of tasks being scheduled. Each task has associated with it a column and a row. The elements of the matrix are filled with the task inter-dependencies. The columns contain all the information about what a particular task is dependent on. The rows give information on what is dependent on that task. Three vectors of length N are also part of the SDA. One vector contains the State Objects (SO's). The SO's are simulation packets which keep track of the states of the resources required for a particular task [16]. Another vector has the tasks' execution windows (see section 2.4). The third vector has each task's Current Status Object (CSO), which is used to guide the processing of that task's row in the SDA (see section 2.5).

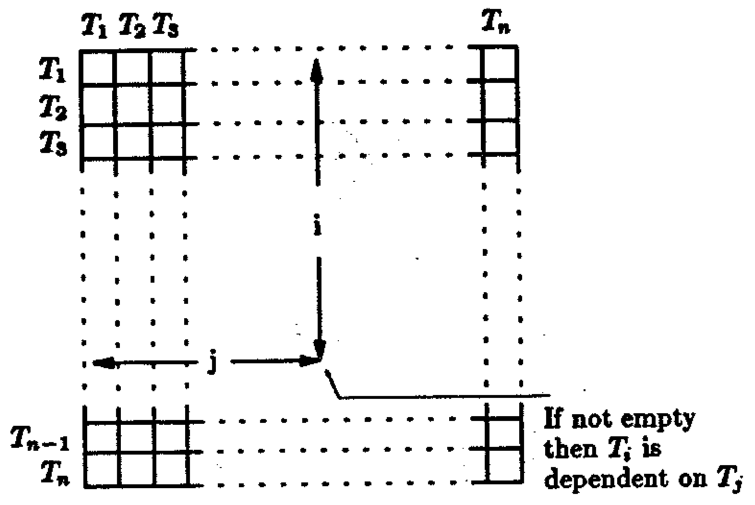


Figure 2: How Dependencies Are Organized in the SDA

With the arrangement described above, the SDA can be used to easily calculate what tasks have had all of their dependencies fullfilled, and how much of a delay, if any, will be necessary before they themselves may be executed. A task has all of its dependencies fulfilled if the column associated with it is empty or has the non-empty elements (i.e., the dependencies) in their final stages of counting down their delays. If the dependency is in the countdown stage then the prerequisite has been fulfilled and it is just a matter of some temporal delay being exhausted before the task may be executed. If some element $SDA_{i,j}$ is not empty nor in the countdown stage then that indicates that task j is still dependent and waiting for task i to be put on the schedule before it may follow suit.

When a task is put into the schedule the SDA makes it a simple job to start all of the task's dependencies counting down. If task i is added into the schedule then its CSO is appropriately altered. The CSO then sends an appropriate message to the task's row in the SDA. The message is passed down all of the dependencies in that row instructing them to go into the countdown state — a state where the delays between tasks start to decay.

When the SDA is being updated to take into account some amount of simulated time that has gone by, the updating process is done row by row. At each row the appropriate CSO is checked to see whether or not the dependencies in that row are in the countdown state. If they are, then an appropriate amount of time is deducted from each of the delays in that row. Relative deadline counters are also decremented. If one of the deadline counters should go negative then a *deadline* critic (see section 2.7.3) is created. All of the execution windows in the execution-window vector are updated when the SDA matrix is updated. The execution windows can be updated independently of the status in the CSOs.

When the *ready tasks*, those whose dependencies have all been fulfilled, are being searched for, the algorithm is: find all of the columns in the SDA whose elements are empty or in the countdown stage. For each task, whose column meets these requirements, check its SO, found in the vector of SOs, and check to see if there exists a legal state transition to that state required for the task. If a legal transition can be found then that task is passed onto the SDA's *thinning* routines. The transition time for the expansion is matched against that task's closest deadline (both absolute and relative). If the transition time is found to be longer than the time scheduled before the task's deadline would occur, an *impending-deadline* critic is formed.

In large problems with few ordering and/or resource constraints it is possible that there will be a great many *ready tasks*. The information available to the SDA, after calculating the ready tasks, is sufficient to thin down the number of possible expansions passed back to the rest of the scheduler. The SDA selects just the union of the following tasks:

- tasks with other tasks dependent on them
- the two tasks with the nearest deadline
- the two tasks with the smallest state-transition time
- the two tasks with the least time left over when their delay times are subtracted from their nearest deadline.

These particular criteria are used in order to select the tasks that are needed to avoid resource and/or dealine violations. Tasks with dependencies are always included to ensure that a relatively unconstrained task that has heavily constrained dependents is carried along far enough in the search process to at least be evaluated by the final rating heuristics (see section 2.7). Each of the above criteria has its mark placed in a data area for the tasks to which it applies. When a task is incorporated into a schedule prefix, its marks can be examined to ascertain why it was placed at that particular point in the schedule.

The SDA performs one final evaluation of the ready tasks that are left. It is quite possible that the ready task with the nearest deadline has its deadline closer than the time it would take to perform the state transition for some of the other ready tasks. If one of the tasks with a lengthy state transition should be used as the prefix expansion then a deadline violation would arise when the SDA is next updated. In order to avoid this, the SDA evaluates the remaining ready tasks, and calls a *potential-deadline* critic to eliminate those expansions that would cause such a violation.

2.2.3 Determining if a Feasible Schedule Can be found

The dependency matrix of the SDA can be thought of being an adjacency matrix [1] for the dependency graph of the problem being scheduled. The dependency graph for a scheduling problem must not contain any cycles if a feasible schedule is to be produced. The SDA offers a fast and efficient way to check if any cycles exist. The process for this is similar to the methods used to reduce a determinant in linear algebra.

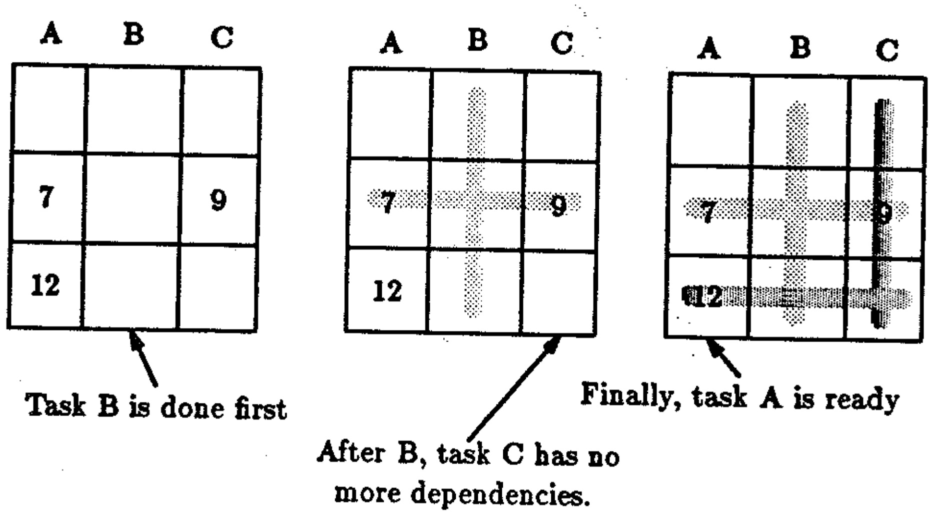


Figure 3: Reducing a Three Task SDA for Schedule Validation

The initial SDA, that is formed from the original task specifications, must have at least one column with every element empty or in countdown mode. If this were not the case then that would mean that every task in the problem had a prerequisite task in that problem. If every task is dependent on some other task in the problem, then none of the tasks can ever be done. The column that is empty of prerequisite objects, and the row with the same index, can be deleted from the SDA (see figure 3). After the deletion operation one or more columns must become similarly *clear*. At each iteration all of the empty or countdown-only columns are deleted, and with them their associated rows. If the complete SDA (not counting the rows and columns containing the SOs, CSOs, and execution windows) is eventually deleted, then there are no contradictory or mutually exclusive ordering constraints present in the problem.

2.3 Task Orderings and Prerequisites

In real planning situations it is very common for the various tasks to be interrelated in a variety of ways. The most typical of these relations is that one task is required to precede another. For most Noah-type planners [19] this is the only type of explicit relation that two tasks can have.

However, when time is brought into the picture, additional inter-task relations can occur.

These include:

- delays between tasks
- deadlines between tasks
- disjunctive sets of tasks
- conjunctive sets of tasks.

Delays between tasks are most common when two tasks are the start and finish processes of some higher level task. For example, if $task_1$ is to put a steak into the broiler and $task_2$ is to take the steak out, then $task_2$ must come at least ten minutes after $task_1$.

Deadlines between tasks often crop up in the same situations. If the person designing the task specifications does not want to risk eating a steak that has been burned beyond recognition, then it would be a good idea to put a deadline between $task_1$ and $task_2$ of about fifteen minutes. Deadlines between tasks allow the scheduler to avoid searching through the possible schedules that, in this example, would cause the steak to be removed from the broiler more than fifteen minutes after it went in. Possible schedules that have a larger delay between tasks than that specified by the inter-task deadlines cause deadline violations, and are disqualified immediately. The structure representing inter-task deadlines must be set up so that a deadline violation occurs by the point when the scheduler's time has advanced past the deadline. Waiting until the task with the deadline is put into the schedule to realize that a deadline violation has occurred could be very detrimental to the efficiency of the scheduler as a whole.

In the HTS, inter-task delays and deadlines are represented for each pair of tasks by a *Prerequisite Object* (hereafter referred to as a PO). A PO is associated with the dependent task (in the steak cooking example the dependent task would be taking the steak out of the broiler, i.e. $task_2$). Through the machinations of the SDA (see section 2.2.2) the PO is notified when the prerequisite task is scheduled. Once the prerequisite is scheduled, the PO starts counting down its delay and deadline counters — keeping pace with the scheduler's clock. When the delay has counted all the way to zero, the PO sends out a message that, as far as that particular prerequisite task is concerned, the dependent task is ready to be scheduled.

As the scheduler's clock continues to count down, the PO's deadline counter also continues to count down. If the deadline counter reaches zero before the dependent task has been put into the scheduler, then the PO sends a message to the SDA saying that the current schedule has failed.

Of course a PO need not have both a delay and a deadline. Some tasks and/or domains (such as the blocksworld) have no deadlines, just ordering constraints.

Disjunctive task situations (e.g., not eating just before, during, or right after working out at the gym) are handled by having two deactivated POs (one associated with each task) each making its task dependent on the other. Each PO has the appropriate ordering delay so that if, for example, eating lunch is put onto the schedule then going to the gym cannot be scheduled to happen less than an hour later. When one of the tasks is put onto the schedule then an appropriate message is sent down that task's row of the SDA; this message causes the correct PO to be activated. This ensures that the two tasks will not be scheduled to come too close together in time.

Conjunctive situations (e.g., schedule all your job interviews within a week of each other) can be handled in a similar manner. In this case, all the POs associated with each task are set up to be triggered by every other task in the conjunctive set. Each of these POs would have an appropriate deadline (in this case a week). Thus once the first of the tasks was scheduled all of the other tasks in the set would be under a deadline to be scheduled quickly.

2.4 Execution Windows

Tasks often have absolute temporal constraints that must be met in addition to the ordering constraints describe in the previous section.

It is not at all uncommon to have tasks constrained by the particular time at which they are supposed to be executed. Most of the activities in a person's day are constrained in this way. When one thinks of an itinerary, usually a time-mapped schedule like that in Figure 4 is pictured. However, such a mapping into time is quite ambiguous. For example, what does **12:00-1:00 lunch** really mean? Does lunch start at precisely noon and terminate exactly one hour later? Or perhaps lunch occurs sometime between noon and 1:00pm. If the latter is the case then how long does the actual event of **lunch** last? In a similar vein, does Figure 4 actually suggest that one is to leave work at 5:00 pm, or is that just the earliest time one could leave. Perhaps 5:00 pm is a deadline one must leave the office by; at 5:00 pm vicious guard dogs are released into the building.

What is lacking in the figure is a sense of *slop* or *fuzziness*. A preliminary task description should not specify exactly when anything should happen. *Ideal times*, as suggested in Deviser [23], try to specify the exact time at which a task should be done, but by the time other scheduling considerations come into play the ideal times must usually be ignored. What first appears, in isolation, as a good time for an event to take place may not look as attractive when brought into context with the other tasks that must be scheduled. Hence tasks in the HTS are assigned *execution windows* which represent the total interval over which the task may be done. The POs can then be used to control the length of the actual task. The inter-task dependency delays capture the fuzziness of the length of the task, while the execution windows on the endpoints allow the necessary slop for the task's placement

6:45 get up

8:00 go to work

12:00-1:00 lunch

5:00 go home

. . .

Figure 4: An Itinerary of a Typical Day

in the time line. In this way the start and end points of a task are represented by intervals; the duration of the task is controlled by the countdown and deadline encoded in the PO linking the start and end subtasks.

Unfortunately not all tasks have a continuous interval over which they can be executed. When trying to get an appointment it is not at all uncommon for the window to be described as something like: "Come in between 9am and noon or between 2 and 5pm." However, even in cases such as this there is an interval over which the task must be executed, i.e., between 9am and 5pm. It just so happens that there is a two hour interval during which the task may not be accomplished. There is a fundamental difference between trying to schedule the task at 1pm and trying to put it at 6pm; for the first, there will be an additional hour delay reported by the task's window, for the latter, placement of the task in the schedule at 6pm will cause a deadline violation.

Thus an execution window is more than just a window, it is actually a series of non-overlapping windows in which the task can be done. The execution window is implemented similarly to a PO. The difference is that when the end of a window is reached, the delay until the next window begins is reported, and a deadline violation is not set up. When the scheduler's simulation time has passed the end of all of the windows, a deadline violation is created if the associated task has yet to be scheduled.

2.5 Current Status Objects

There can be up to $N^2 - N$ task interdependencies — delays and deadlines, for a set of N tasks. In order to make scheduling a smooth and efficient process, the system requires some method to track which of these dependencies are active and which are not. This is an especially complex problem in the case of disjunctive tasks, since the dependencies may switch about depending on how the schedule is being expanded.

To handle the task interdependency bookkeeping the scheduler uses a set of N *Current Status Objects*, further referred to as CSOs. A CSO is created for each task. The main purpose of a CSO is to alert the scheduler to which of the following states its task is in. These states include:

- *waiting*
- *schedulable*
- *finished*
- *looping*
- *evaporated*
- *failed*

When most tasks first start out they are in a *waiting* state. This means that the task is dependent upon some other task or set of tasks that have yet to be scheduled. When a task's CSO reports that the task is waiting then no further consideration is given to that task on that iteration of the scheduling process. A task's CSO is computed to be in the waiting state during the preprocessing done before the scheduling process ever begins. Such a CSO can only be moved into another state by the tasks upon which it is dependent.

After all of the tasks upon which a particular task is dependent are scheduled then that task's CSO reports that it is now in the *schedulable* state. Schedulable means that a task is not dependent upon any other task directly. It is possible that such a task may still not be worked into the schedule due to a resource conflict or other state restriction. Only tasks in the schedulable state are checked for state restrictions and transition delays. Those that have no such restrictions will be returned to the scheduling routines as possible extensions to the partial schedule which is currently being expanded. They are the *ready tasks.*

When one of the *schedulable* tasks is placed onto the current prefix then its state is changed to *finished.* Being in the finished state indicates that this task need not be examined or updated again. When a CSO becomes *finished* it sends out a series of messages to the CSOs that are still in the waiting, schedulable, or looping states informing them of its change in status.

It is also possible for the scheduling of one task to cause another task to *evaporate* — i.e., be removed from the list of schedulable tasks. For example, if a message must be sent to Tom there may be several plans for accomplishing it:

1. Go to Tom's house and deliver it to him personally
2. Write and mail him a letter
3. Go to a computer and send him electronic mail

All of these are workable plans, and it may be very difficult if not impossible to tell which tactics will become the most expedient to actually execute when they are being examined at plan expansion time. Only as the HTS creates a plan is it possible to determine which tactics, for sending a message to Tom, would be the best to use. Therefore all of the plans are submitted to the scheduler with a note that if any of these plans are incorporated into the schedule, then the other two should evaporate (i.e., their CSOs should report a status of *evaporated*). Upon evaporation any tasks formerly dependent on the evaporated task are released from those dependencies.

Disjunctive sets of tasks, where all of the tasks will eventually be used in the schedule, are implemented in a similar manner. When a task, which is the start of one of the disjunct intervals, is worked into the schedule, its CSO passes a special message to the CSOs of all tasks which are not to overlap its interval. This message instructs the CSOs to activate the necessary POs in the SDA to ensure that the disjunctive intervals cannot overlap.

A status of *looping* has the combined effects of the finished and schedulable statuses. The looping status indicates that the task has been placed onto the schedule at least once but must be placed onto the schedule at least one more time. The POs dependent on a looping task start to countdown much as if the task had been finished, but the looping task remains active to be updated by the SDA as necessary. It is the CSO that eventually changes the status of the task from *looping* to *finished* when a sufficient number of iterations have been incorporated into the schedule. What this number is depends on the exact type of task. In a repeated task such as: **wash both the cars**, the number is known at plan construction time. The number 2 is placed in the repeated task's CSO. A task such as: **hit the nail until it is all the way in** would have a maximum estimate of hammer strikes placed in the CSO. An observation task would have to be set up as part of the loop to change the task status to *finished* should the nail be knocked in all the way before the planned number of strikes had been executed.

Whenever an intertask deadline violation occurs, a message is sent to the task's CSO. In such a case the CSO reports a status of *failed*. This serves as a flag to the scheduling routines and causes the partial schedule under consideration to be discarded. The particular task which caused that failure is kept track of for debugging purposes.

2.6 Scheduling Windows

The output of the HTS is an *itinerary* where an itinerary is a list of tasks and events, and the windows and states under which they should take place. The states described in the output are those defined by the various task's SOs [17], as worked out by their state transitions.

The windows produced by the scheduler differ in several aspects from the execution windows described in section 2.4. Most notable among the differences is that the scheduling windows are made up out of three time values rather than two. These three times are:

- The *lower* time bound
- The *safe* time bound
- The *upper* time bound

Each of these times represents a different aspect of the schedule with regard to the execution time of the task. The times can also serve to aid in execution monitoring.

The schedule is guaranteed to succeed if the *lower* bound is followed. This bound is calculated using the minimum time spent for accomplishing state transitions and the minimum execution time for each task upto that point in the schedule. The lower bound refers to the earliest time that can be planned on to start a task, assuming that all the tasks preceding this point have taken as little time as possible.

The *safe* time bound, coming out of a scheduling window for a particular task, represents the last time that the task can be started and have the schedule remain valid. If a task is executed at a time later than the *safe* time then it is impossible for the remainder of the tasks to be executed without a deadline violation occuring.[1] The *safe* time limit is calculated from the total ordering by propagating the *lower* limit backwards from the various deadlines incorporated in the schedule. The *safe* times are similar to *latest allowable dates* in PERT, and serve a similar function.

Finally, the scheduler also produces a pessimistic view of the execution world. In this case the scheduler uses the *upper* bounds on the tasks' execution times and state transitions. The longest times for a task's execution time and state transition are added onto the minimum of the previous task's *safe* and *upper* bounds.

Just because execution times remain below the *upper* bounds specified in the schedule does not have any bearing on whether or not the execution is proceeding in a manner that concurs with the schedule. *Upper* bounds serve only as warnings as to how long a task or transition could possibly take; only the *safe* limit can be used to monitor whether or not a schedule is remaining valid. Places where the *safe* limit comes before the *upper* time bound indicate places where the schedule is likely to fail. These are the prime spots for the insertion of monitor tasks and contingency plans.

[1]This of course assumes that one of the tasks remaining to be executed does not take substantially less time to be executed than was marked in the task specification. Similarly for the amount of time spent executing the state transitions.

2.7 Schedule Rating Heuristics

While the algorithms for scheduling tasks have been covered in detail, there still remains the question of how the schedule searcher decides which prefix, and which expansion for that prefix, the system should be working on. The answer to that question is that the system maintains a rating of how promising each prefix and expansion is. What that rating is based on and how it is calculated make up the remainder of this section.

2.7.1 Rating the Schedule So Far

At any point in the scheduling process, it is easy to find out how much time and how many tasks have been scheduled for a particular prefix. With these numbers it is trivial to calculate the average time scheduled for each task. Assuming that the average task's time remains constant throughout the remainder of the scheduling process allows the system to estimate the total time required for the schedule, if the number of tasks to be scheduled is known. Unfortunately, the number of tasks to be scheduled is often not well known when only a part of the schedule has been decided upon. The number of iterations for some loops may yet to have been calculated. If decisions on exclusive disjunct task sets have not been scheduled then they also can contribute to the task number uncertainty. However, estimates can be made of the number of tasks remaining to be scheduled, and the average task time can be improved upon by sampling the delay times stored in the SDA. These can help to improve the quality of the *Estimated Total Time*, or ETT for short.

The ETT can give some estimate of the quality of its own value by tracking its change during the scheduling process. If two prefixes' ETTs are the same, but one prefix's ETT has decreased every time its prefix has been expanded, then that prefix should get a higher score (since its ETT has a history of being pessimistic). So the score associated with an ETT is a function of both the value of the ETT and the trend of the ETT over the last several prefix expansions.

The ETT's reliability (and likewise its overall weight in the prefix's rating) depends heavily on the proportion of the ETT that has already been scheduled. In other words, a prefix that is almost a complete schedule should have its ETT play a more important role than a prefix that has only a couple of steps in it.

The score based on the schedule-so-far is a function of how much has been scheduled and how much the system thinks there is yet to schedule. Additionally, the rating is modified by whether the estimates it is based on are believed to be conservative or optimistic. If the estimates are believed to be conservative (based on the trends of the ETT) then the score associated with it would be raised; if optimistic then the score would be lowered. The rating based on these time estimates helps to distinguish markedly different prefixes from one another.

2.7.2 Rating Possible Prefix Expansions

Choosing the best prefix expansion for a particular prefix relies on information that is local to the possible expansions, and therefore mostly provided by the SDA. The particular factors that play a role in the rating of expansions are:

- the expansion's closest deadline
- the expansion's state-transition delay
- the expansion's prerequisite delays
- the tasks that are dependent on the task in the expansion
- the task's place in the overall set of state transitions.

The way that these factors interact with one another can also effect the expansion's score.

If a task is placed onto the schedule at a time such that its prerequisite delays will run out simultaneously with its state-transition delays, then the robot will spend little or no time idle. The task will assuredly be done before any deadline violation can occur. If the task is scheduled so that its prerequisite delays are larger than the transition delays then the robot will spend that difference being idle — a sign of an inefficient schedule. This is wasted time, and is detracted from an expansion's score.

In most scheduling problems a better solution can be found if the scheduler has freedom on how to order the tasks to be performed. One set of factors that limit the scheduler's freedom are the prerequisite constraints. Prerequisite constraints can keep a large percentage of the tasks unavailable for scheduling if their prerequisites have yet to be fulfilled. For creating a near optimal schedule, and for meeting certain deadlines, it is desirable to schedule the prerequisites as early as possible in a schedule. Therefore, the number of tasks for which a particular expansion's task serves as a prerequisite has some effect on that expansion's score.

The relationship between an expansion's state-transition delay and that task's nearest deadline is an important factor in an expansion's score. If the state-transition delay takes up almost all of the time before the task's deadline, then that expansion has a higher priority than an expansion that has a large gap between its nearest deadline and transition delay. The reason for this is quite simple: an expansion that contains little slack between the end of the state-transition and the task's deadline can probably not afford to be delayed any longer. Those expansions with a wide disparity between their deadlines and transition delays have more leeway, and are more likely to be postponable without an adverse effect. Expansions whose state-transition delays take up the exact amount of time before the task's deadline comes up form a *delay-deadline* critic, to ensure that they are tried as the next, and usually only (see section 2.7.3), expansion.

State-transition delays for different tasks can fluctuate greatly depending on the order in which the tasks are done. To take this into account the *resource tour time*, or RTT, is figured into each prefix's overall rating. An RTT can be thought of as the minimum total state-transition delays for the tasks yet to be scheduled. If there were: ten tasks: **A**, **B**, ... **J** left to be scheduled, each task had to be done at a different workstation, the only source of state-transition delays was moving the robot from one workstation to the next, and the robot was currently at location **R** then the RTT for task **A** would be equal to:

$$\frac{\mathsf{tsm}(A : B, C, D, E, F, G, H, I, J) + \left|\overline{RA}\right|}{10}.$$

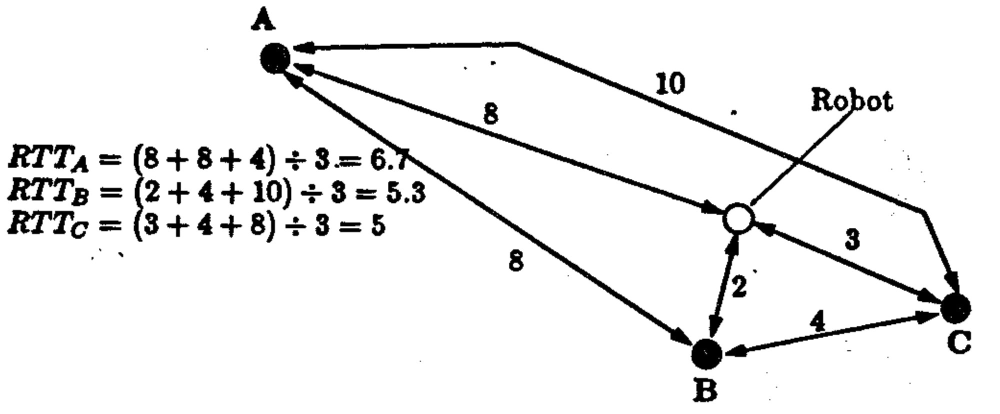
The function **tsm** calculates the traveling salesman tour for the ten workstations with the constraint that workstation A must be the first visited. The RTT for task **B** would involve a traveling salesman tour that was constrained to visit workstation B first. Unlike traditional traveling salesman problems, the distances between *cities* are neither commutative nor associative, due to the nature of state transitions. It is possible that not all RTTs are possible due to resource conflicts among the tasks.[2]

An RTT is calculated for each prefix expansion with the constraint that the first state to be visited is the state associated with the task to be added to the schedule by that expansion. The other stops on the tour are the other ready tasks. Figure 5 shows the RTT scores for three schedulable tasks whose only resource is the robot's position. For this simple example each task's RTT reduces to calculating the shortest route for visiting every workstation starting with the one needed for the task. The result is then normalized. The figure gives the RTTs for each of the three tasks.

To make the claculation of RTTs tractable, a traveling salesman approximation should be used — rather than calculating the optimal path. The algorithm used in this implementation is nearest neighbor insertion. This runs in time $O(M^2)$, but since M is only the ready tasks, this does not significantly add to the overall scheduling complexity.

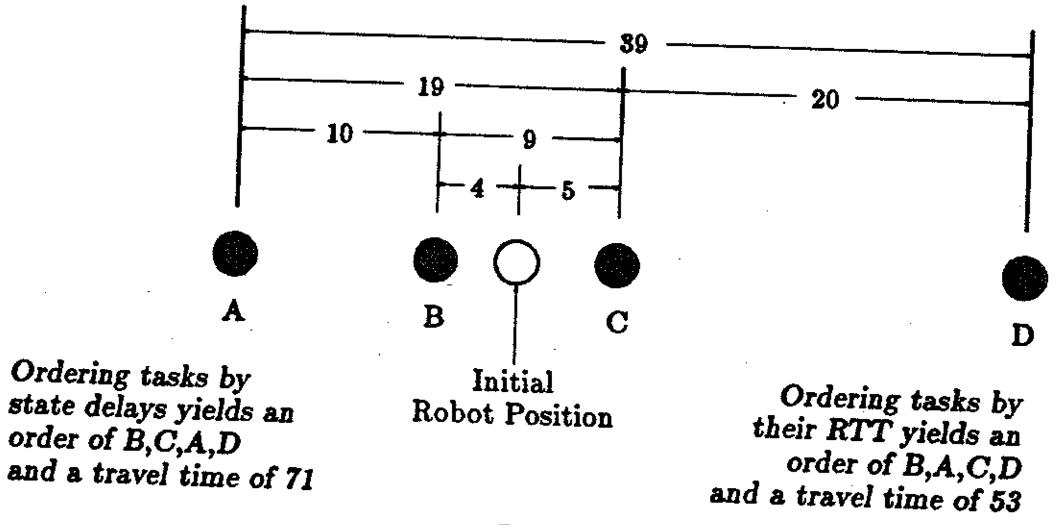
The RTTs are used for scoring purposes, rather than the individual transition delays, because the RTTs are immune to certain types of situations where scoring only on transition delays would produce a very bad schedule. A trivial example of a situation where RTTs produce better scores that transition delays is shown in Figure 6. The figure presents a problem where the robot must visit four workstations: A, B, C, and D. The workstations are positioned so that ordering visits based upon the shortest transition delay will actually produce the most time-consuming schedule. The RTTs suggest a schedule that is considerably more efficient. Additional resources can transform this problem into one with similar behaviour in several dimensions.

[2] A resource conflict among the ready-tasks is not indicative of a serious resource conflict. It is quite likely that a task that was not yet ready, or was thinned from the list of ready-tasks, will supply the necessary resources so that all tasks can be scheduled.



A
10
Robot
8
$RTT_A = (8+8+4) \div 3 = 6.7$
$RTT_B = (2+4+10) \div 3 = 5.3$
$RTT_C = (3+4+8) \div 3 = 5$
8
3
2
4
C
B

Figure 5: Calculating Three RTTs for Three Ready-Tasks



39
19
20
10
9
4
5
A
B
C
D
Ordering tasks by state delays yields an order of B,C,A,D and a travel time of 71
Initial Robot Position
Ordering tasks by their RTT yields an order of B,A,C,D and a travel time of 53

Figure 6: Why RTTs Are Used Instead of Transition Delays

2.7.3 Final Scoring, Re-Scoring, and Critics

The factors that go into rating a prefix and expansion have been explored in the sections immediately above, but the question remains: How are these factors combined into a final score? The answer to that question is not as simple as one might hope.

Different scheduling problems have different scoring needs. A relatively lightly constrained problem can have an optimal score produced by paying close attention to state-transition delays, waste time, and getting prerequisite tasks scheduled as quickly as possible.

Deadline-intensive sets of tasks need more attention paid to minimizing waste time and the relationship between prefix expansion deadlines and their state-transition delays, in order to just find an answer. Resource-intensive tasks must pay more attention to domain-specific scoring factors, and must make sure to get prerequisite tasks scheduled as early as possible.

Unfortunately, many problems have part of their schedules being unconstrained, while other parts are resource and/or deadline intensive. Since there is no one scoring system that works for all problems, the HTS uses a flexible system which adapts itself to the problem being worked on.

Scoring factors are initially balanced towards producing a maximally efficient schedule. As the scheduling process proceeds, the system may try to expand a prefix that leads to a dead end. When that dead end in encountered it is analyzed and an appropriate critic is constructed. A dead end can be one of three types:

Update Deadline: a failure that is spotted by the SDA when the time required for the most recently scheduled task is added into the SDA and one of the tasks remaining to be scheduled is forced past its deadline. A *Deadline* critic is created in response to this failure.

Impending Deadline: a failure that is uncovered when the SDA is comparing ready tasks' deadlines to their state and prerequisite delays. An *Impending Deadline* critic is created to handle these situations.

Resource: a failure that results because of there being no legal state transition from the prefix's last state to any of those states of the tasks that are otherwise ready. A *Resource* critic is made up in these circumstances.

These types of dead ends, and their associated critics, are used to reset the balance in the score of schedule prefixes and their possible expansions. The ways in which the scoring are changed are shown in Table 1. The critics then go through each of the prefixes and possible expansions in the search front, and rescore them.

Heuristic	Absolute Deadlines	Relative Deadlines	Resource Restrictions
Task Weight	+	-	+
Deadline	+	+	
RTT	+	-	
Domain			+
No Wait		-	-
Waste Time			-

Table 1: Scoring Changes Due to Scheduling Dead Ends

Each time a dead end is encountered the scoring is altered. The extent of the alteration is proportional to the size of the dead end. If a deadline is missed by a couple of minutes, the chages to the scoring weights are much smaller than if it suddenly comes to light that five tasks will each go several days overdue.

The effects of the scoring changes are cumulative, but not long lasting. If several consecutive impending deadline failures are encountered, then the scoring is going to be heavily shifted in order to avoid future failures of that type. However, it is quite possible that the part of the schedule that is likely to cause difficulties in that direction has been or will very soon be completely scheduled. In order for the scheduler to produce efficient schedules the scoring strategy must revert back to the efficiency-directed strategey with which the system started out. Scoring changes decay over time. The rate of the decay is affected by the number and spacing of the dead ends that are encountered. If several somewhat closely spaced dead ends of the same type are encountered then the decay rate is reduced. If no dead ends are encountered for a substantial period then decay rate increases in an attempt to revert the HTS to an efficiency-oriented scheduling system.

In order to reduce the number of failures encountered of the various deadline varieties, two other critics are commonly constructed:

Potential Deadline: a critic used when some of the ready tasks have state transitions that are longer than the nearest deadlines of some of the other ready tasks.

Delay Deadline: a critic that is produced when a ready task's transition delay takes exactly as long as the time scheduled before its nearest deadline.

Neither of these critics causes the mass re-scoring that is initiated by the other critics described above. Instead, these critics eliminate some of the expansions for the prefix. The Potential Deadline Critic removes all of the expansions for a prefix that have state transitions taking longer than the time until the nearest deadline of any of the expansions. This is to avoid the Update Deadline failure that would certainly result if one of the expansions eliminated were to be used. The Delay Deadline critic removes all of the prefix expansions, except for the one that cause the critic's appearance. Since that task's state-transition delay will take up all of the time before its deadline, it must be scheduled next — otherwise it will surely violate its deadline. The other possible expansions are saved for the somewhat rare cases where the scheduling of one of them would cause the task that formed the critic to evaporate.

The flexible scoring system outlined above, when combined with critics and and the structured search provided by the SDA, usually leads to efficient schedules produced with a near minimum of search. The efficiency of a schedule is measured by the total scheduled time required for achieving the tasks while meeting all of the task constraints. Because of the heuristic nature of the search algorithm, a formal proof of the system's effectivenss is difficult to provide. However, extensive testing has been performed on a wide variety of tasks and problem sizes. The system has always found a solution to the problem where one was known to exist (and sometimes where no solution was initially believed to exist). These solutions have always matched or beaten any solution worked out by hand. Additionally, the system has reliably performed in linear time, with a coefficient of approximately 3.5. For details on the testing procedure, the reader is refered to [17].

3 Strategic/Tactical Planning with the HTS

In this section the role the HTS plays in planning is discussed. The HTS has been integrated into several AI planning systems. They are described below. At the end of this section, the HTS is compared to more traditional OR project planning techniques.

3.1 Integrating the HTS Into Planning

The HTS has been used in several different planning systems. In the Bumpers planner [17] the HTS coordinated all planning activity. The planning domain was the coordination of sensory resources with motor activities in a mobile robot. A primary function of the HTS was to create feedback loops, and then to integrate those loops with one another to form an efficient plan. Because of uncertainty in the domain, sensory feedback was used to decide when to terminate certain plan steps, and when to go on to the next task.

The Forbin planner [7], [15] explored the effects of various forms of temporal reasoning on the planning process. Forbin is a hierarchical planner which consults the HTS and a temporal data-base, or *time-map*, when doing plan choice. In Forbin, having many alternative plans for a single task is commonplace; tactical plan choice becomes more important in this system then is common in most other planners.

If there are many possible plan choices, then some mechanism must be used to distinguish which is the best choice. In Forbin, this choice is made by the HTS with the help of the temporal data base. First the **T***ime*-**M***ap* **M***anager*, [5], [4] which controls the temporal data-base, is queried with each of the possible plan choices for the task currently being expanded. For each choice, the TMM returns all of the temporal intervals in which it can find no conflict for executing that plan. Associated with each interval and plan, the TMM attaches a list of all of the constraints that executing that plan, at that time, would add to the state of the world. If there is no time at which a particular plan choice could be successfully executed, the TMM returns no intervals.

The TMM makes all of its predictions based on information formed from the partial order of the *plan prefix* (i.e., the plan as it has so far been constructed). Therefore, if the TMM finds a reason for believing that a task would not execute successfully, it is almost certainly correct. However, a recommendation by the TMM to execute a specific plan during a specific temporal interval does not necessarily guarantee success. The TMM is not capable of detecting adverse plan interactions that are dependent on the exact order in which the plan steps are executed. To make the final plan choice, and to guarantee that a choice with a high probability of success does exist, the plans, intervals, and their constraints are sent to the HTS.

The HTS gets, from the time-map, a copy of the plan prefix. It then attempts to create a schedule of plan-steps using the prefix, and one of the plan choices for the task being expanded. Which plan choice is picked depends on which the scheduler says will produce the best schedule overall. Several or all of the choices may be examined before one is finally picked.

The schedule that is produced by the scheduler is consistent with all of the constraints that were already introduced by the plan as previously constructed. Additionally, all of the constraints added by the TMM for the plan and time choice picked by the scheduler are also taken into account. The tactical plan used in the HTS's schedule becomes the plan for the task being expanded. It will become incorporated into the larger plan. The particular time-slot chosen, along with the rest of the time and ordering constraints added by the scheduler, will not be enforced, in keeping with the policy of least commitment.

Work is currently underway to extend the Forbin planner through the exlicit separation of strategic and tactical plans. This new planner (whose architecture is shown in Figure 7) has a separate plan library for strategic planning and for tactical planning. The strategic planning is done after the style of Forbin; a hierarchical least-commitment planner forms the basic plan outline, which is then criticized, verified, and has final plan selection made by the TMM and HTS.

Once this basic strategic plan[3] has been constructed, its steps are fed to the robots who will carry out the plan. The robots then feed back their sensory impressions to the temporal database allowing the planner to monitor the plan's execution. Should the actual execution deviate, beyond acceptable limits, from the plan, then the reactive tactical planner part of the system will be called into action. This part of the planner uses an associative memory index to find relevant repair tactics for the type of deviation occuring and the plan strategy that it is affecting. The repair routine is then given to the HTS, along with the unexecuted remains of the original plan, in order to produce a coordinated repair for the plan. The repaired plan is then sent to the robots and further execution is monitored similarly.

3.2 The HTS Verses PERT for Project Planning

The HTS bears many similarities in form and function to organizational techniques such as PERT (see survey in [10]). Both have similar point-based representations for tasks (in

[3]While this plan is referred to as "strategic" it contains many tactical decisions (such as the selection of which particular version of a plan to use, and various timing decisions) that are missing from planners such as Noah

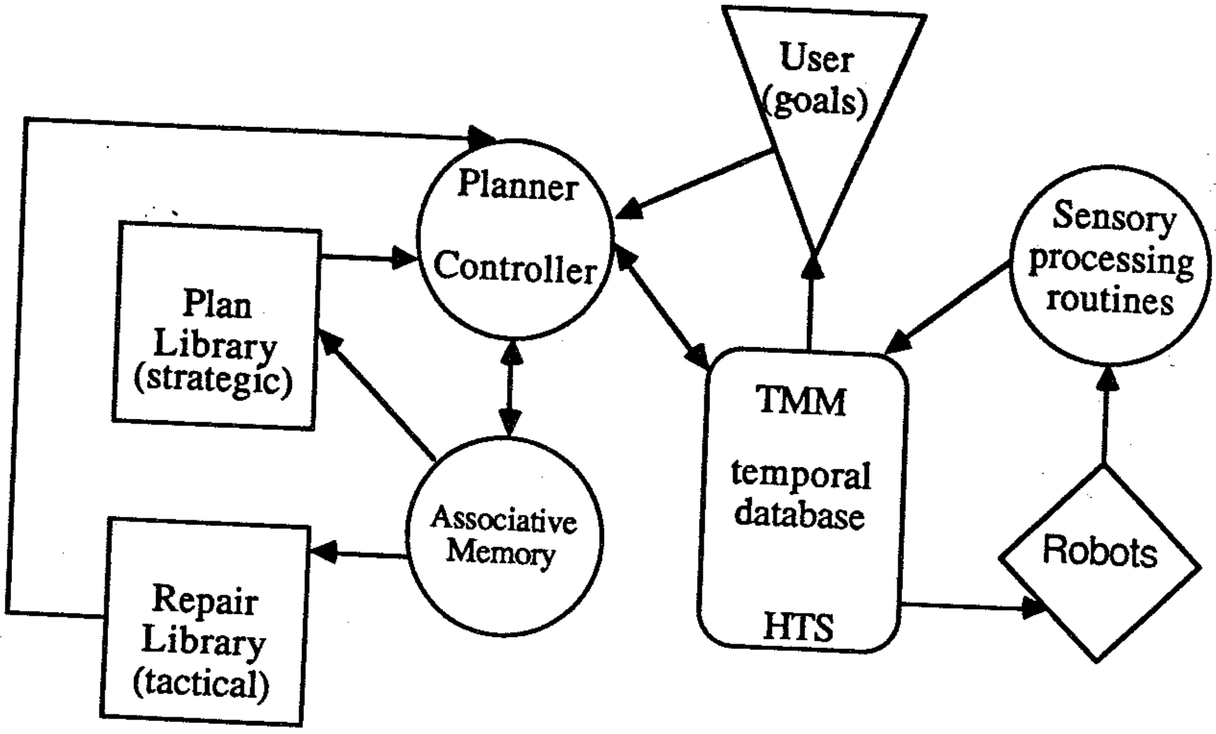


Figure 7: The Architecture for the Forbin Strategic/Tactical Planner

PERT called events): tasks take no time — rather they represent the start or conclusion of some activity. In fact, the HTS creates and uses a matrix representation of something very much like a PERT network in its SDA.

However, PERT leaves a much heavier organizational burden on a project organizer than does the HTS. Figure 8 shows a simple PERT chart for a task that involves creating two machined objects: a widget and a gizmo. It would not be uncommon for each of the two tracks of the PERT chart to be put together by different people. The chief widget administrator might draw up the top part of the chart and a gizmo expert might draw up the bottom. The gizmo part of the chart may be created up without the knowledge that a widget chart is also being drawn. It is possible that there may only be one lathe — and each adminstrator is expecting to use it towards the completion of his part of the project. In PERT it would be up to the project organizer to make sure that adequate resources were available for each subproject. The HTS would automatically allocate the resources available.

If there was only one lathe available then the project manager might rearrange the chart

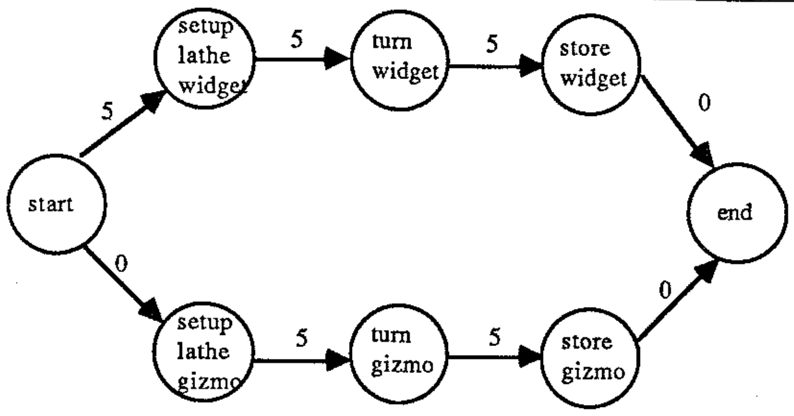


Figure 8: PERT Chart for Creating a Widget and Gizmo

as shown in Figure 9. However, the time estimate of 0 for the lathe setup for producing a gizmo may have been based on the state of the lathe at *start*, which will have changed now that a widget has been produced. If the lathe steps had to be linearized and the setup time for a gizmo increases if the lathe has been used for a widget, then the HTS would have automatically scheduled the gizmo operations first. Additionally, if a single worker could carry both the finished widget and gizmo, the two storage steps would have been automatically collapsed, by the HTS, into a single trip. This is beneficial since it does not increase the overall project time, and it conserves resources (a workers time and energy).

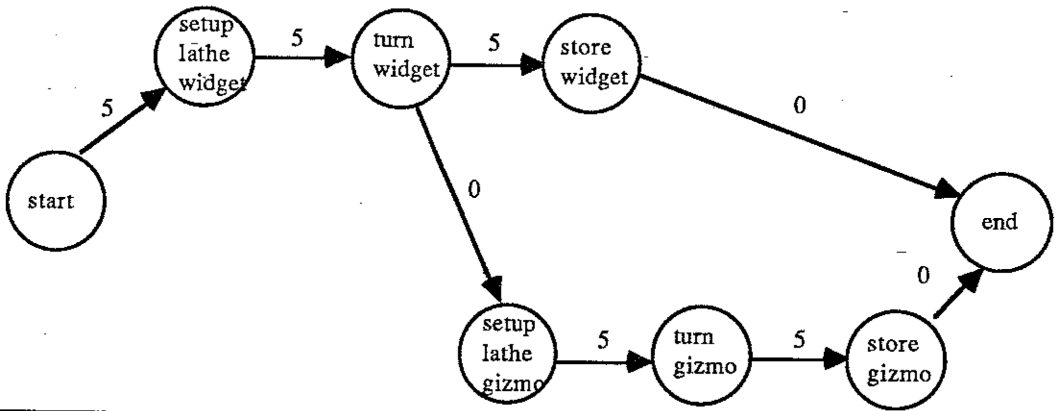


Figure 9: Revised PERT Chart for a Single Lathe

An additional advantage of the HTS is that it integrates into an overall planning system. The HTS can select among alternate plans for accomplishing a subproect. The selection is made upon the basis of which plan fits in most smoothly with the other elements of the overall project. Alternate plans for a single task do not fit in well with the PERT paradigm since PERT looks at each event as an individual rather than as a piece of a sub-project as does the HTS (or a Gantt chart [11]).

4 Summary and Conclusions

Previous popular planning systems such as Noah and Nonlin have concentrated on deriving and debugging high-level strategic plans. The plans produced by such systems are seldom adequate for real-world domains because the hierarchical least-commitment planning strategy avoids tactical decision making issues such as uncertainty, temporal constraints, and resource allocation/constraints.

The algorithm for a heuristic task scheduling module has been outlined. This program has been implemented in NISP [13], a dialect of Lisp, and integrated into several planning systems — giving them expanded abilities to handle uncertainty, temporal and resource constraints, and the ability to reason about repeated tasks. Additionally, the scheduling module provides a global view of the plan, and a detailed simulation of the plan — which provides a much greater confidence in the accuracy of the system.

The simulation of the plan provided by the task-scheduling module allows more detailed tactical decisions to be made during the planning process. This means that the input to the system can be simpler while still deriving more detailed results than traditional task scheduling methods such as PERT. During plan execution, the task scheduler can be used to coordinate reactive changes made to the plan — to counter any unexpected deviations from the orginal schedule. The efficient search through the space of totally-ordered plans is necessary for effective tactical planning, and catching context-dependent interactions at the strategic planning level.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

[2] James Allen.
Maintaining knowledge about temporal intervals.
Comm. ACM, 26(11):832–843, 1983.

[3] David Chapman.
Planning for Conjunctive Goals.
Technical Report TR-802, MIT AI Laboratory, 1985.

[4] Thomas Dean.
Temporal Imagery: An Approach to Reasoning about Time for Planning and Problem Solving.
Technical Report CSD/RR 433, Yale University, Department of Computer Science, October 1985.
PhD thesis.

[5] Thomas Dean.
Temporal reasoning with metric constraints.
In *Proc. CSCSI 84*, pages 28–32, Canadian Society for Computational Studies of Intelligence, 1984.

[6] Richard Fikes and Nils J. Nilsson.
Strips: a new approach to the application of theorem proving to problem solving.
Artificial Intelligence, 2:189–208, 1971.

[7] R.J. Firby, T. Dean, and D. Miller.
Efficient robot planning with deadlines and travel time.
In *Proc. of the 6th Int. Symp. on Robotics and Automation*, pages 97–101, IASTED, Santa Barbara, CA, May 1985.

[8] M.S. Fox, B. Allen, and G. Strohm.
Job-shop scheduling: an investigation in constraint-directed reasoning.
In *Proc. of the National Conf. on Artificial Intelligence*, pages 155–158, AAAI, Pittsburg, PN, August 1982.

[9] Steve Hanks and Drew McDermott.
Default reasoning, nonmontonic logics, and the frame problem.
In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 328–333, Philadelphia, PN, August 1986.

[10] F. S. Hiller and G. J Lieberman.
Operations Research.
Holden-Day, 1974.

[11] R. I. Levin and C. A. Kirkpatrick.
Planning and Control with PERT/CPM.
McGraw Hill, 1966.

[12] B. Lowerre.
The HARPY Speech Recognition System.
PhD thesis, Carnegie-Mellon University, 1976.

[13] Drew McDermott.
The NISP Manual.
Technical Report 274, Yale University Computer Science Department, June 1983.

[14] Drew V. McDermott.
A temporal logic for reasoning about processes and plans.
Cognitive Science, 6:101–155, 1982.

[15] D. Miller, R.J. Firby, and T. Dean.
Deadlines, travel time, and robot problem solving.
In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1052–1054, IJCAI, AAAI, Los Angeles, CA, August 1985.

[16] David Miller.
Ai planners for shop scheduling.
In *Proc. of the Second Annual Artificial Intelligence and Advanced Computer Technology Conference*, pages 220–225, 1986.

[17] David Miller.
Planning by Search Through Simulations.
Technical Report 423, Yale University Computer Science Department, October 1985.
PhD thesis.

[18] David Miller.
Scheduling Heuristics for Problem Solvers.
Technical Report 264, Yale University Department of Computer Science, 1983.

[19] Earl Sacerdoti.
A Structure for Plans and Behavior.
American Elsevier Publishing Company, Inc., 1977.

[20] Earl D. Sacerdoti.
Planning in a heirarchy of abstraction spaces.
Artificial Intelligence, 5:115–135, 1974.

[21] Austin Tate.
Generating project networks.
In *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, pages 888–893, IJCAI, Cambridge, Ma, U.S.A, August 1977.

[22] Steven Vere.
Temporal scope of assertions and window cutoff.
1984.
JPL, AI Research Group Memo.

[23] Steven A. Vere.
Planning in time: windows and durations for activites and goals.
IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-5(3):246–267, May 1983.

[24] Robert Wilenskey.
Planning and Understanding.
Addison-Wesley, 1983.