AI Magazine Volume 15 Number 2 (1994) (© AAAI)

A Structured View of Real-Time Problem Solving

Jay K. Strosnider and C. J. Paul

■ Real-time problem solving is not only reasoning about time, it is also reasoning in time. This ability is becoming increasingly critical in systems that monitor and control complex processes in semiautonomous, ill-structured, real-world environments. Many techniques, mostly ad hoc, have been developed in both the real-time community and the AI community for solving problems within time constraints. However, a coherent, holistic picture does not exist. This article is an attempt to step back from the details and examine the entire issue of real-time problem solving from first principles. We examine the degrees of freedom available in structuring the problem space and the search process to reduce problem-solving variations and produce satisficing solutions within the time available. This structured approach aids in understanding and sorting out the relevance and utility of different real-time problem-solving techniques.

As computers become more ubiquitous, they are increasingly being used in applications that sense the environment and directly influence it through action. Such applications are subject to the real-time constraints of the environments in which they operate. These systems, which have deadlines on their processing requirements, are referred to as real-time systems. Their functional correctness depends not only on the logical correctness of the results but also on the timeliness. These systems include manufacturing, control, transportation, aerospace, robotics, and military systems.

Historically, these systems were relatively simple, operating in well-characterized environments. However, the emerging generation of increasingly sophisticated real-time systems will be required to deal with complex, incompletely specified, dynamic environments in an increasingly autonomous fashion (Laffey et al. 1988). To do so requires that time-constrained problem-solving techniques be incorporated into real-time systems. According to the problem-space hypothesis posited by Newell and Simon (1972), *problem solving* is defined as search in a problem space. They define a *problem space* as a set of states and a set of operators linking one state with the next. A *problem instance* is a problem space with an initial state and a set of goal states. Extending this notion, we define *real-time problem solving* as time-constrained search in a problem space.

In the field of cognitive psychology, a study of air traffic controllers conducted by the Rand Corporation is the first known attempt to characterize human problem-solving behavior when subject to real-time constraints (Chapman et al. 1959). In the field of AI, the roots of real-time problem solving can be traced to the development of search-based programs to play chess, with time constraints being placed on each move. Current techniques have evolved from these early studies and applications.

Real-time systems tend to be critical in nature, where the impact of failures can have serious consequences. Before the next generation of real-time problem-solving systems can be deployed in the real world, it is necessary to be able to predict the performance of the system and make statements about the system's ability to meet timing constraints. There is currently no coherent theory of real-time problem solving. The existing real-time problem-solving systems are, at best, coincidentally real time (Laffey et al. 1988).

The problem of building real-time problem-solving systems has been approached by both the real-time community and the AI community. The real-time community has taken a system approach, integrating AI tasks into existing real-time systems. The AI community has concentrated primarily on developing techniques and architectures to facilitate time-constrained problem solving. A number of techniques have been developed in both the real-time community and the AI community, each with their respective terminology. A study of the current literature leaves one confused about techniques and terminology. It is also not clear how the techniques from the two research communities can be combined in solving a given problem.

This article attempts to sort out the terminology and the techniques developed in the two communities and provides a common, integrated approach to real-time problem solving. The basic premise of this article is that a structured approach to real-time problem solving exists based on the fundamental degrees of freedom available in using domain knowledge to structure the problem space and the search process to produce best-so-far solutions in limited time. Existing real-time problem-solving techniques are shown to be compositions of these fundamental degrees of freedom.

This article is organized as follows: First, The Problem provides a background on the current state of the art in real-time problem solving and introduces some of the existing real-time problem-solving techniques. The second section discusses the relationships between deadlines, computation time, and solution quality. The sections entitled A Structured Approach and Knowledge Retrieval introduce our structured approach. Mapping Existing Techniques examines related work in real-time problem solving and the kinds of problem tackled. The final section explores the limitations of this work and proposes future research.

The Problem

The challenge with real-time problem-solving tasks is that the worst-case execution time is often unknown or is much larger than the average-case execution time. Although conventional real-time tasks have execution-time variations because of data dependencies, real-time problem-solving tasks have additional variations because of search and backtracking (Paul et al. 1991). A blind application of conventional real-time scheduling theory to real-time problem-solving tasks results in systems that either cannot be scheduled or have low-average resource utilization. To tackle this problem, we examine the fundamental differences between conventional real-time tasks and problem-solving tasks and discuss how these differences affect the execution-time variations of these tasks.

First, let us consider the execution-time variations of conventional real-time tasks. Typical real-time signal-processing algorithms have little to no variation associated with their execution times: Regardless of the complexity and size of most signal-processing algorithms (for example, fast Fourier transforms, filters), generally no data dependencies can cause the execution times to vary. The input data are simply processed in a uniform, deterministic fashion. However, control-oriented, real-time tasks often have data dependencies. As the system to be controlled increases in complexity, the number of data dependencies will likely increase, resulting in increased variations in the execution time of real-time tasks.

Next, let us consider the execution-time variations of problem solving in the context of the problem-space hypothesis advocated by Newell and Simon (1972). Figure 1 illustrates a continuum of tasks that range from knowledge poor to knowledge rich. On the far left, knowledge-rich tasks are fully characterized, and an explicit algorithm exists that transforms a given set of input into an appropriate output. There is no notion of search or backtracking at this end of the spectrum. Variations in execution time are associated solely with data dependencies, as is the case for conventional real-time tasks.

As one moves to the right, either the task characteristics or their interactions with the environment are not completely known. Heuristics are now required to search the state space for an appropriate result. At the far right, there is no knowledge to direct the search, resulting in a blind search. In this case, one would expect to have a large variation in execution time. Most potential applications fall between the two extremes shown in figure 1. As one moves back to the left, increasing knowledge can be applied to reduce the variations caused by search.

Observations: 1. Execution-time variations because of data dependency and execution-time variations as a result of search and backtracking are orthogonal. 2. The combination of execution-time variations because of data dependency and search results in real-time problem-solving tasks that tend to have much larger execution-time variations than conventional real-time tasks. 3. Execution-time variation is a true measure of the quality

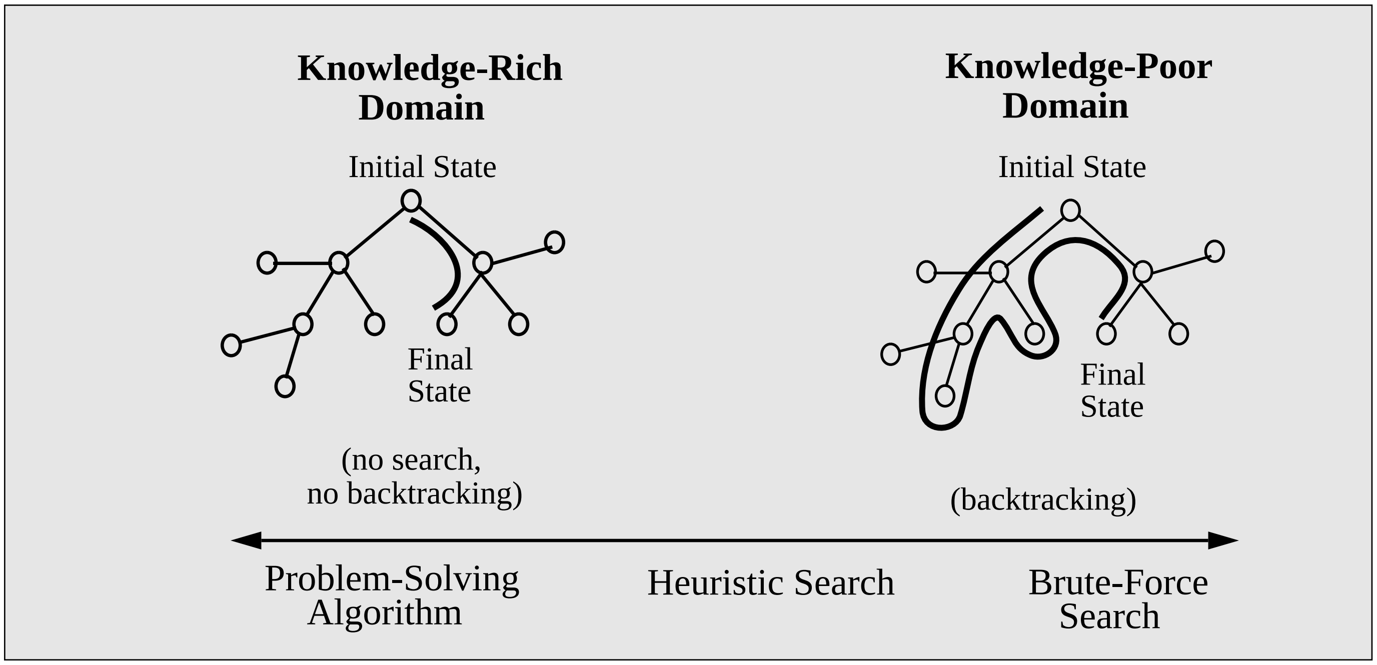


Figure 1. The Search Spectrum.

of domain knowledge available and the efficiency of its use. (If a particular application is knowledge rich and still has a lot of execution-time variations, then it is a poorly constructed system.)

In the discussion that follows, whenever we refer to search, we mean generalized search with backtracking (see side bar). Other forms of search are referred to as iterative algorithms. Generalized search occurs in a problem space. Although some domains have natural problem spaces (chess, for example), most real-world problems are made perversely difficult because one has to discover how to treat them as search problems in the first place. Then the initial representation (initial problem space, the operators, the start state, and the goal state) is derived from the problem specification.

Throughout the remainder of this article, we use an application example to illustrate the abstract ideas we present. The application example we consider is a car-tracking system. In large cities such as New York, antitheft devices now include beacons installed in cars, which allow the police to track down a stolen car. The tracker has a display that indicates the approximate direction and range to the target stolen car. For any given combination of current tracker location and current target location, there is no procedural way to determine the path from the tracker to the target. The solution requires search. Further, the stolen vehicle is being driven by an intelligent person in an unpredictable fashion. This dynamic aspect of the example problem is common to most interesting real-time problem-solving domains. For this application, the search space is the set of all road intersections, represented as a graph with edges corresponding to road segments. The operators correspond to the intersections encountered in daily life: turn right, turn left, go straight ahead, turn around. The direction and range information can be used as a heuristic function to guide the search process.

Deadlines, Computation Time, and Solution Quality

In this section, we explore the origin of time constraints in real-time problem solving and the way that these time constraints translate into deadlines. We highlight the effect of execution-time variations of real-time problem-solving tasks on the determination of system timing correctness using real-time scheduling theory. We then discuss how the required computation times, deadlines, and available computation time determine the solution quality that can be guaranteed.

Deadlines

Time constraints in real-time problem solving can manifest themselves in a variety of ways (Dodhiawala et al. 1989; Laffey 1988; Stankovic 1988). They can be either static or dynamic. In conventional real-time systems, deadlines are known time quantities and are generally directly related to environmental constraints (Wensley et al. 1978). In our car-tracker example, tasks associated with engine control, automatic braking, and so on, are conventional real-time tasks with hard deadlines directly tied to the physics and dynamics of the underlying environment. The execution properties of these tasks are well behaved, and the deadlines tend to be constant and known.

In other cases, the deadlines of tasks can be dynamic. In our example application, the deadline of the real-time problem-solving tracking task is derived from the requirement that the tracker proceed at normal speed limits at all times. Thus, as the tracker moves between states (intersections) in the problem space, he/she must determine which operator (right, left, and so on) to execute before he/she gets to the next intersection. The deadlines of these real-time problem-solving path-planning tasks are dynamically variable, depending on the vehicle speed and the distance to the next intersection. Note that the dynamic nature of the stolen car precludes the tracker from path planning far into the future (multiple intersections ahead).

Another source of deadlines in the example application would be at the system level, such as levying a requirement that the tracker must capture the stolen car within 20 minutes. Although such deadlines are easily levied, it is difficult or impossible to guarantee their being satisfied in highly dynamic environments.

More often, a real-time problem-solving task must share the computing resource not only with other real-time problem-solving tasks but also with conventional real-time tasks.

Computation Time

If a real-time problem-solving task were the only task on the computer, the computation time available to the task would be the entire computation time from the current time to the deadline. More often, a real-time problem-solving task must share the computing resource not only with other real-time problem-solving tasks but also with conventional real-time tasks. In the application example, the real-time problem-solving task associated with finding a path to the stolen vehicle might have to share the central processing unit with vehicle control functions that have tight real-time constraints. Further, if we extend the tracker problem to support tracking of multiple stolen cars simultaneously, then multiple real-time problem-solving tasks and multiple conventional real-time tasks would have to coexist on the same resource.

There are two aspects to the problem: (1) determining what computation time is required to satisfy the functions of each task and (2) determining whether the set of tasks can be scheduled such that each task's computation-time requirements can be satisfied before its deadline.

Determining the computation time required to support conventional real-time tasks is generally straightforward. Determining the computation time required to support real-time problem-solving tasks highlights the fundamental problem of scheduling them, that is, their potentially large execution-time variations. For well-structured and well-understood applications rich in domain knowledge, accurate estimates can be obtained. As the application becomes less well characterized, the execution-time estimates become increasingly pessimistic. For highly dynamic and unstructured applications, reasonable estimates might not even be possible.

Given the worst-case execution times of a set of tasks, real-time scheduling theory can then be applied to determine whether the deadlines of the tasks will be met. One such approach is the *earliest deadline scheduling approach* (Liu and Layland 1973), which orders task scheduling by deadline. According to the Liu and Layland equation, a real-time task set scheduled using the earliest deadline scheduling algorithm is guaranteed to meet all deadlines if

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} + \frac{C_1^{ai}}{T_1^{ai}} + \ldots \frac{C_m^{ai}}{T_m^{ai}} \leq 1$$

where the C_i's represent the worst-case computation times of a set of n conventional real-time tasks, and C_j^{ai}'s represent the worst-case computation times of a set of m periodic real-time problem-solving tasks. The T_i's represent the periods of the tasks (deadlines are assumed to be the same as the periods). The larger the variations between the average-case execution time and the worst-case execution time, the more pessimistic the value of C_j^{ai} is and the lower the average-case resource utilization (Paul et al. 1991).

An introduction to real-time scheduling theory can be found in Sha and Goodman (1990) and Sprunt (1989). For a summary of fixed-priority scheduling in real-time systems, see VanTilburg and Koob (1991). For recent work on dynamic priority scheduling, see Schwan and Zou (1992) and Jeffay, Stanat, and Martel (1991).

Solution Quality

Let us now compare the computation time available with the computational requirements of the task. If the computation time available is greater than the worst-case execution time of the task, the task can be guaranteed to generate the optimal solution before its deadline for all valid input in its design space. Because real-time problem-solving tasks have large execution-time variations as a result of search, it is often not possible to guarantee that the time available for computation would be greater than the worst-case execution time of the task (Paul et al. 1991). If the computation time available is less than the worst-case execution time, we could either declare the task set unable to be scheduled or attempt to generate acceptable solutions of lower quality (satisficing solutions).

Satisficing solution: A *satisficing solution* is one whose solution quality is greater than an acceptable threshold. The notion of a satisficing solution places no restriction on the manner in which the solution is generated or on the evolution of the solution quality as a function of time. A satisficing solution necessarily implies a compromise in solution quality.

The task can be guaranteed to generate satisficing solutions if the worst-case time required to generate satisficing solutions for all valid input in the design space is less than the computation time available. If there is insufficient computation time to guarantee satisficing solutions for all valid input or if the deadline itself is unknown, the only option is to structure the search space and the search process to generate best-so-far solutions.

Best-so-far: A *best-so-far solution* defines a search evolution that generates intermediate results whose **expected** solution quality improves monotonically. We note that the property of best-so-far evolution is independent of generating satisficing solutions because satisficing solutions can be generated by processes that do not produce any useful intermediate results.

The next section examines the degrees of freedom and shows how existing real-time problem-solving techniques are compositions of the fundamental degrees of freedom.

A Structured Approach

When attacking any complex real-time system problem, conventional or AI, it is imperative to structure the solution strategy in such a way as to achieve the required response-time goals. To deal with the complexity of such systems, the overall problem is decomposed into a set of subproblems. We assume that this process of decomposition is applied recursively until we end up with a collection of basic subproblems, some of which are procedural, and some of which are search based. The basic subproblems indicated here are the smallest computational entities that can be treated independently (that is, can produce a result and be scheduled as an independent computational entity).

Any representation of a problem forms an abstraction of the actual problem. Moving from one level of abstraction to another involves a change in the number of characteristics considered. Higher levels of abstraction space look at progressively fewer but relatively important characteristics. Techniques such as ordered monotonic refinement (Knoblock 1990) are useful in developing abstraction hierarchies. Abstraction hierarchies at the problem level guide the partitioning of a problem into subproblems. The following discussion assumes a specific definition of subproblem that we use in this article.

Techniques such as ordered monotonic refinement (Knoblock 1990) are useful in developing abstraction hierarchies.

Subproblem: Each search-based subproblem operates at a single level of abstraction and has a search space with an initial state, a goal state, and some number of intermediate states connected by operators.

The single level of abstraction constraint enables us to make unambiguous assertions about the timing properties of the various techniques addressed later in this article. It also allows us to regard the transition from one level of abstraction to another as moving from one subproblem to the next. Real-time search issues are dealt with at this level. Because search involves selecting and applying the operators until the solution is found, the deadlines at this level fall into two categories: (1) a deadline to solve the search-based subproblem and (2) intermediate deadlines to select and apply operators during the search process.

We define a problem and its relation to subproblems as follows:

Problem: A *problem* is a set of subproblems connected by a set of intersubproblem operators that trigger subproblem transitions between one subproblem and the next. These transitions occur when the goal state of a subproblem is reached.

We use the problem level to discuss issues relating to multiple subproblems, transitions between subproblems, the scheduling and executing of different subproblems in sequence or in parallel, and the mixing of procedural and search-based subproblems. The timing constraints at this level are to solve the end-to-end problem.

In large application domains, this definition of problems and subproblems can be generalized to arbitrary levels of nesting. In dynamic environments, it might be necessary to determine the decomposition of the problem into subproblems at run time. However, for a particular problem instance, the assumption of a subproblem being at a single level of abstraction is not a restrictive assumption.

We now develop a framework for real-time problem solving by identifying the degrees of freedom available in structuring the problem space and the search process to provide solutions of increasing quality as a function of time. We then evaluate the effect of this approach on the execution-time variations of the problem-solving tasks and illustrate this effect using qualitative examples. This work was motivated by the need for an integrated approach to real-time problem solving in the development of real-world applications (Paul et al. 1992) and extends our initial work in reducing problem-solving variations to improve the timing predictability of real-time AI systems (Paul et al. 1991). Additional details of the structured approach can be found in Paul (1993).

Our structured approach manages the large execution-time variations of real-time problem solving in two ways. First, we try to reduce the execution-time variations by using domain knowledge to reduce the maximum number of states searched in the worst case. After having reduced the number of states searched in the worst case, if the residual variations are still large or if the deadline itself is unknown, the only option left is to structure the search space and the search process to produce solutions of improving quality as a function of time (best-so-far property) using the fundamental degrees of freedom.

We postulate that the best one can do in managing problem-solving variations is to use domain knowledge for pruning, ordering, approximating, and scoping. Generic techniques that fall under this framework are as follows: static pruning, dynamic pruning (pruning); best-first heuristic search (ordering); value approximation, function approximation, aggregation (approximating); and scoping in time, scoping in space (scoping).

We find that existing real-time problem-solving strategies fall within this framework and exploit some of these degrees of freedom. Each of these degrees of freedom manifests itself at both the problem level and the subproblem level and has different effects on the worst-case execution time and the ability to generate best-so-far solutions.

Pruning is a fundamental method for reducing the number of states searched. Pruning manifests itself in a number of forms, both during run time and at design time. *Partitioning, problem decomposition,* and *abstraction* are forms of pruning that manifest themselves when setting up and structuring the search space. If a satisficing solution is acceptable, then approximation can be used to reduce the maximum number of states searched in the worst case. *Scoping* determines the fraction of the search space that will be searched and is useful in generating solutions of improving quality with time. *Ordering* determines the sequence in which the states are searched given a search space. We elaborate on these distinctions as we describe each of these techniques in the following subsections.

Pruning

Definition: Pruning is the technique of using domain knowledge to eliminate portions of the search space that are known not to contain the goal state.

Pruning comes in two forms. First is *static pruning* (pruning at design time). Static pruning requires a priori knowledge that certain portions of the search space need not be searched. At the problem level, static pruning manifests itself in the form of partitioning, problem decomposition, and abstraction. Static pruning is used to set up the search space for the search process.

Partitioning is based on the notion that solving a collection of smaller problems is usually better than solving the whole problem in a single step. This divide-and-conquer technique assumes the availability of domain knowledge to break up the original problem into smaller parts and combine the solution of the smaller parts into a coherent solution of the entire problem.

At the problem level, partitioning can be used to divide the problem into subproblems. Run-time indexing is used to a priori determine which parts of the data or knowledge will be applicable in different situations. For example, the tracking problem illustrated in figure 2 can be partitioned into a set of smaller subproblems: finding the path from the current location of the tracker to the highway, finding a highway connecting the

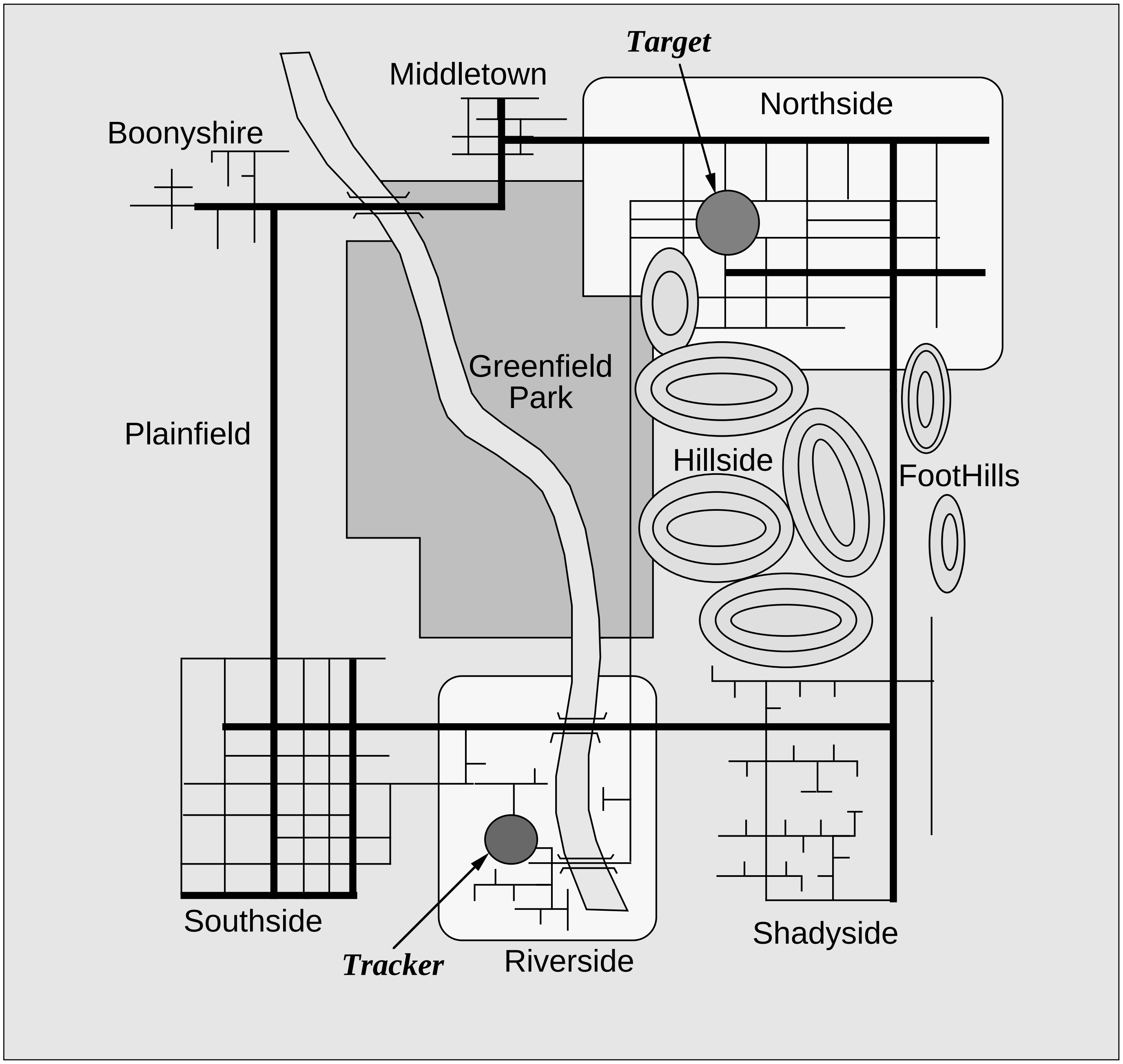


Figure 2. Sample Detailed Map for the Tracker Application.

neighborhood of the tracker to the neighborhood of the target, and finding a path from the highway to the actual location of the target. This problem partitioning is illustrated in figure 3. Within the subproblem of finding a path from the current location of the tracker to the highway, we need only consider those intersections and street segments in the neighborhood of the tracker. Information about intersections, street segments, and traffic flow information from other neighborhoods is not relevant and can be pruned (partitioned) away. The worst-case execution time for this subproblem would then be the time to search through all the intersections and street segments of the particular neighborhood.

The second form of pruning is *dynamic pruning* (pruning at run time). This type of pruning happens during the search process and is based on information that is available only during run time to prune the size of the search space.

Within a subproblem, dynamic pruning is the use of run-time knowledge to eliminate

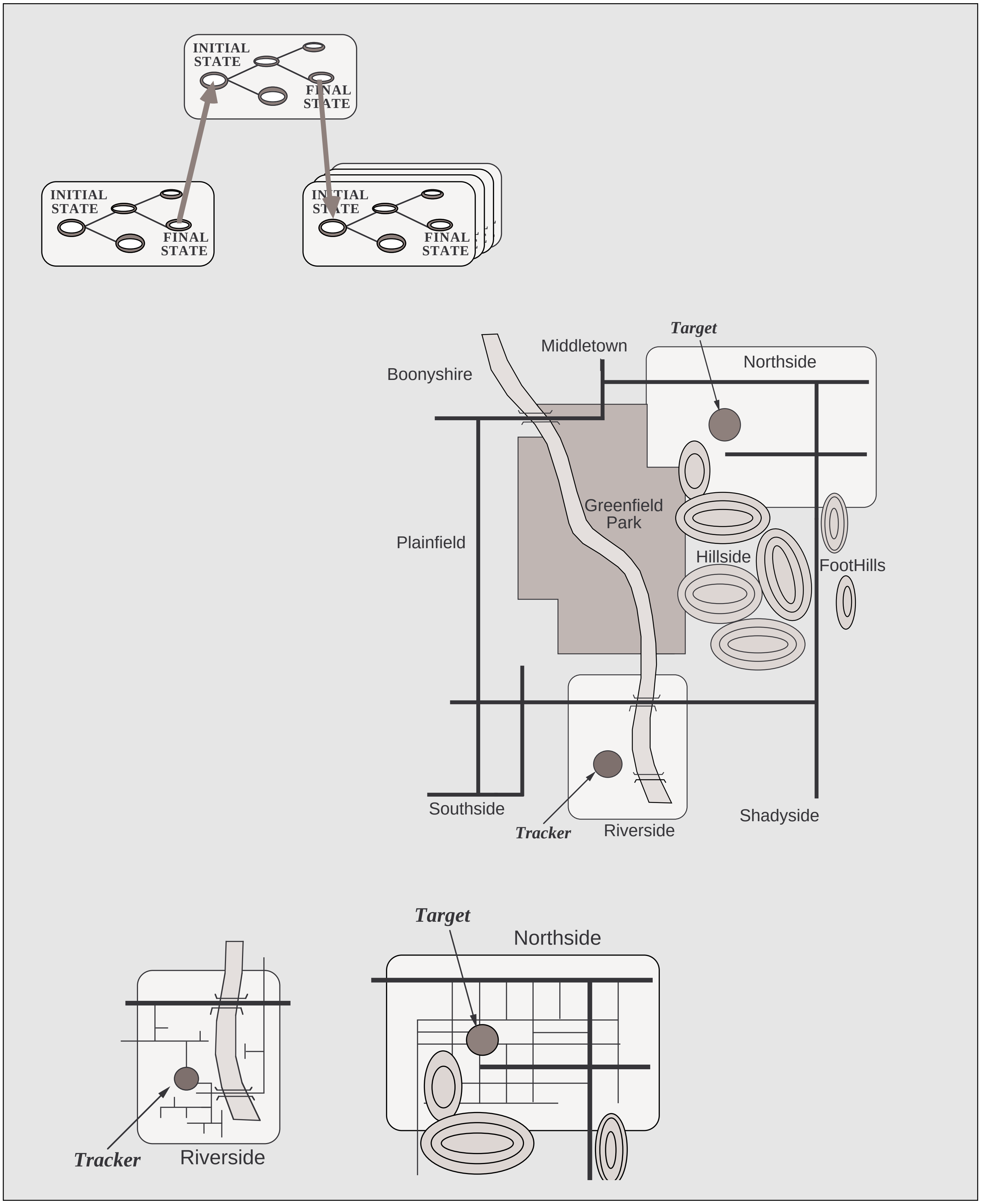


Figure 3. High-Level Problem Decomposition: Generic Case and Application Example.

operators applicable at a given state. In the tracking example, using current traffic information to rule out some of the operator choices would be an example of dynamic pruning. At the problem level, dynamic pruning manifests itself in the deletion of specific subproblems (or their partitions) from consideration for the solution of a particular problem instance.

We bind pruning to the availability of perfect domain knowledge. Hence, pruning using imperfect information or heuristics would not constitute valid pruning in our approach. Subject to these assumptions, pruning (in one of its many forms) is the only way to reduce the worst-case execution time without compromising the goal state or the solution quality.

In the case of static pruning, a priori analysis of the worst-case execution time is possible because the maximum number of states can be estimated at design time. In dynamic pruning, the reduction in the number of states searched is instance dependent. Hence, a priori analysis of the execution-time reduction as a result of dynamic pruning is difficult.

Ordering

Definition: *Ordering* is the technique of looking earlier at the entities (states or subproblems) that are more likely to lie along the solution path.

This type of ordering corresponds to the classical best-first search technique. The A* algorithm (Korf 1987) and most of its derivatives fall into this category, where some form of heuristic function is used to order the evaluation of states. For example, in the tracking application, ordering of operators and states can be done within a subproblem based on the compass heading and the range to the target. At the problem level, the ordering is deterministic: finding a path from the tracker to the main highway; finding a path along the main highways to the neighborhood where the target is; and, finally, finding a path within the target neighborhood to the target. In the more general case, ordering at the problem level is a form of prioritization of evaluation, with higher priority given to those subproblems that are expected to have the highest impact on the overall solution.

The better the search heuristic, the lower the average-case execution time is. We bind ordering to the use of imperfect knowledge. Ordering only changes the probability that a particular state will be visited, evaluated, and expanded. Thus, the worst-case execution time is not affected.

Approximation

Definition: *Approximation* is the technique of reducing the accuracy of a characteristic under consideration.

Approximations at the subproblem level are usually value approximations, but approximations at the problem level tend to be method or function approximations.

Let us assume that the solution to a generic problem can be expressed as

$$\text{Solution} = \phi(a_1, a_2, a_3, \ldots a_n) \ ,$$

where $A = \{a_1, a_2, \ldots a_n\}$ represents the set of characteristics that influence the final solution, and f represents the functional relationship between the characteristics.

Let A_1 represent the set of admissible values for a discrete variable a_1 with cardinality $\eta(A_1)$. In the tracking example, the values of valid speed limits are 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, and 65. Thus, the set of speed limits has a cardinality of 13. Approximation is a technique for reducing the cardinality of the set of admissible values for a variable. (For continuous variables, approximation can be seen to reduce the number of distinct regions in which a continuous variable might lie). In this case, a sample approximation would constitute a mapping from this set of speed limits to values of low speed, medium speed, and high speed. The new approximate set now has a cardinality of only 3. We use the following representation: Let $A_1^{(a)}$ represent the set of admissible approximate values for the variable a_1 such that $\eta(A_1^{(a)}) < \eta(A_1)$. An approximation function is defined from the original set A_1 to the approximate set $A_1^{(a)}$ such that all elements of the original set are mapped to elements in the approximate set.

The approximation function reduces the total number of states $\eta(P_{app})$ that need to be searched in the approximate space P_{app}. Hence, the worst-case execution time of the task is less than the worst-case execution time in the base space:

$$C(P_{app}) < C(P_{base}) \ .$$

With approximations, we note that approximation changes the problem definition; the original problem space, the initial state, the goal state, and the operators are all potentially affected. The implication is that one is willing to accept a less precise goal state.

Approximation methods use different approximate models of the underlying domain, each with a different level of accuracy (figure 4). As the approximation increases, search space size decreases. In the *approximate processing technique* proposed by Lesser, Pavlin, and Durfee (1988), the system esti-

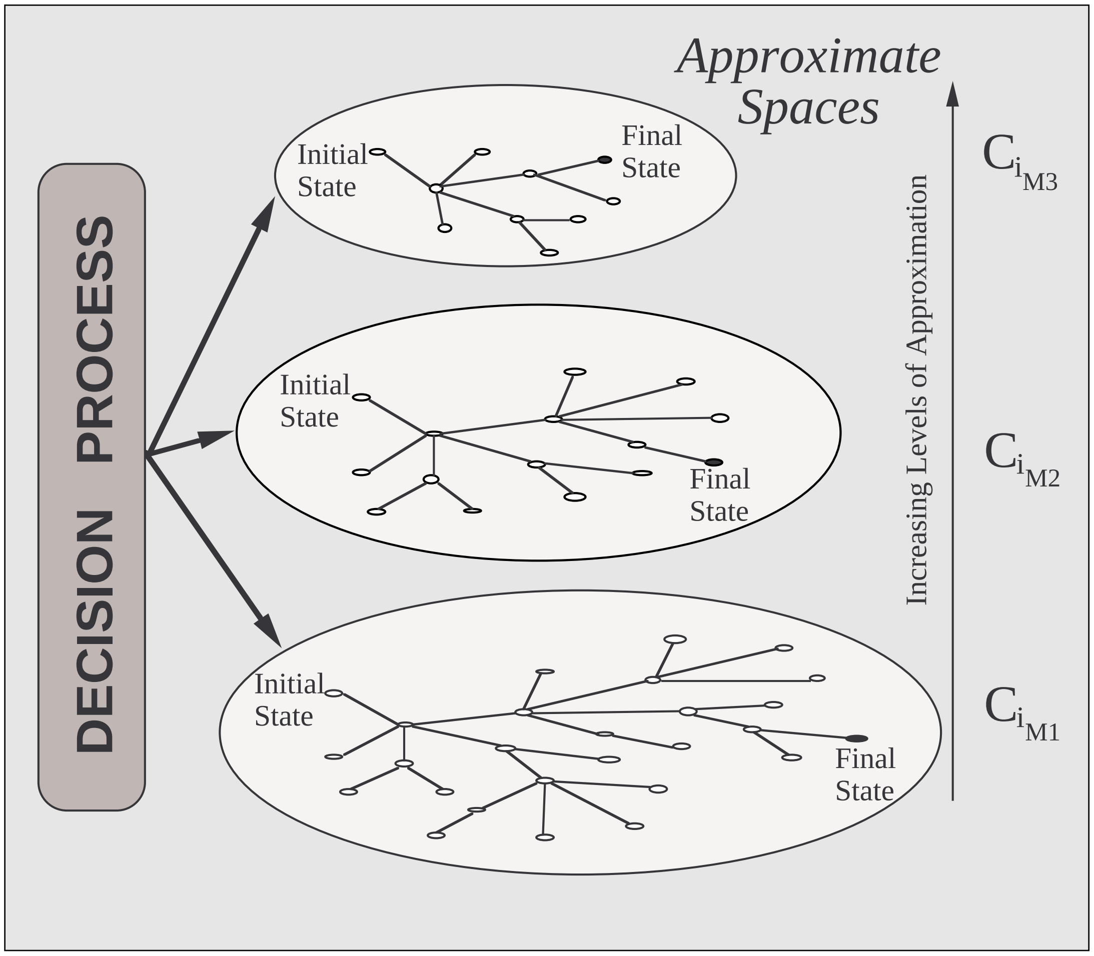


Figure 4. Approximate Models.

mates the amount of time available and uses a decision process to determine the best level of approximation for the time available.
This approach works well as long as the system does not underestimate the amount of time available.

For example, in path planning at the level of roads, using a map of highways only is an approximation. This set of roads has a lesser cardinality than the set of all roads on the detailed map. Note that approximation is reducing the cardinality of the given characteristic (roads).

We also note from the application example that if we use an approximate map for doing the overall planning, then we end up redefining the problem (the problem space, the initial state, and the goal state) to less accurate states. With the approximate map, the best one can do is to get to the vicinity of the target, as shown in figure 5. If the approximate solution is good enough, then approximate processing is acceptable. If the resulting solution is unacceptable, and the intent is to get to the actual location of the target, then approximate processing fails as an end-to-end approach. We need to work with the detailed map in the initial subproblem and the final subproblem to avoid compromising the goal state.

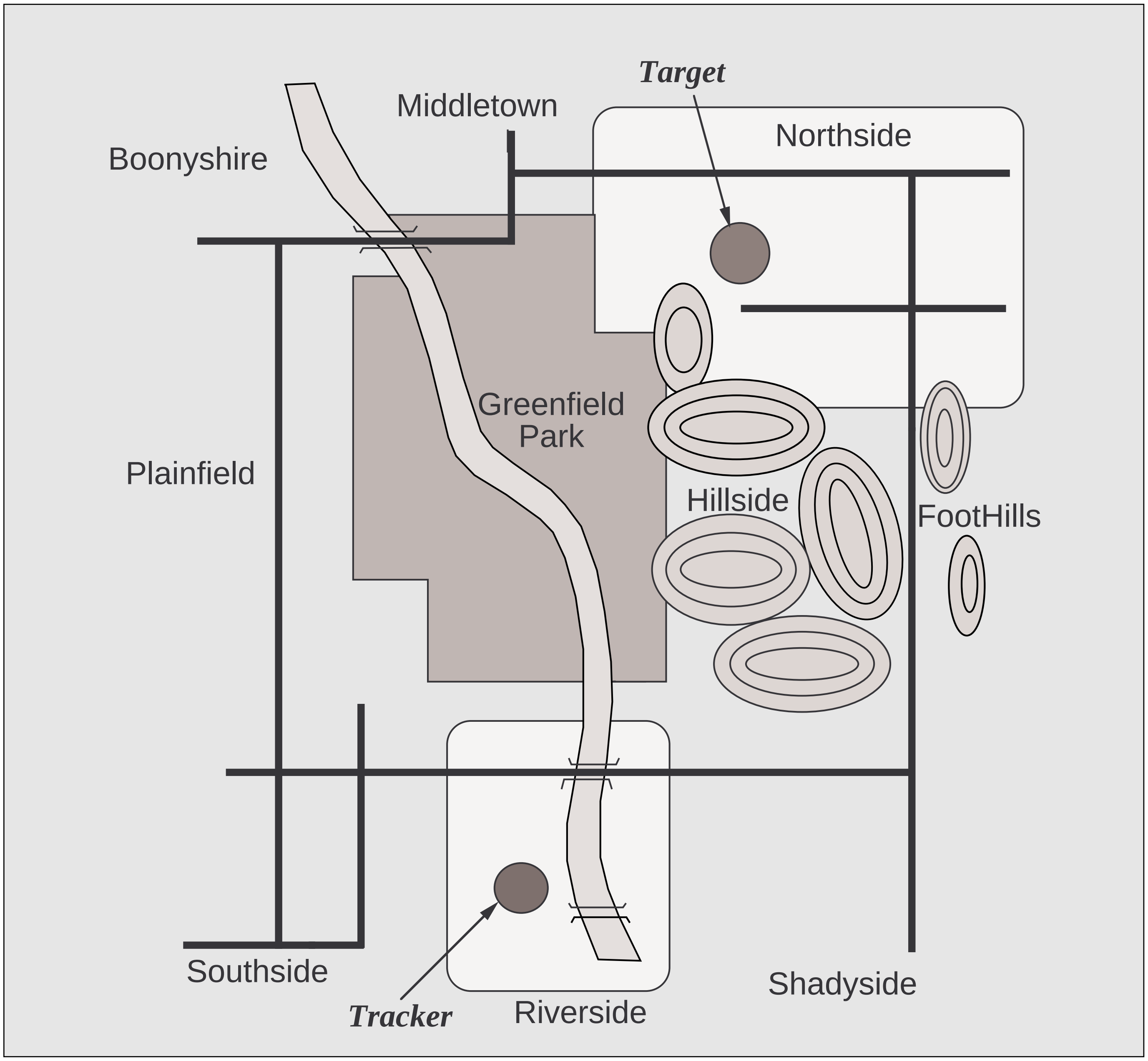
The effects of approximation on execution time can be summarized as follows: (1) the worst-case execution time decreases because the total number of states decreases; (2) the average-case execution time is also expected to decrease; (3) because approximation can involve aggregation, the number of data elements processed is potentially lower; (4) approximation of the evaluation function (by linearization, for example) can reduce its complexity (and the associated state-evaluation time); (5) some approximations allow incremental refinement with additional time, but others require the problem be solved again with a different level of approximation; (6) nonrefinable approximations require us to accurately estimate the amount of time required to generate the solution and then pick the level of approximation that maximizes the expected accuracy for the given allocation of computation time; and (7) the decision process itself needs to be bounded to allow the problem-solving segment enough time to run.

Scoping

Definition: *Scoping* is the technique of controlling (in time and space) the maximum lookahead when choosing the next operator.

Scoping is applicable within a subproblem level in choosing the next move (operator). The initial state and the goal state are not affected; what changes is the manner in which the operators are selected and applied. Figure 6 illustrates scoping in the tracking example. It is possible to trade the increase in scope in range (time) for the increase in scope in azimuth (space). The narrow scope illustrated in the figure allows us to consider lesser information with each move, but we can look ahead more moves (within a fixed-computation time window). The wide scope illustrated in the figure is an example of increased scoping in azimuth (space). Because we consider

Figure 5. An Approximate Map.



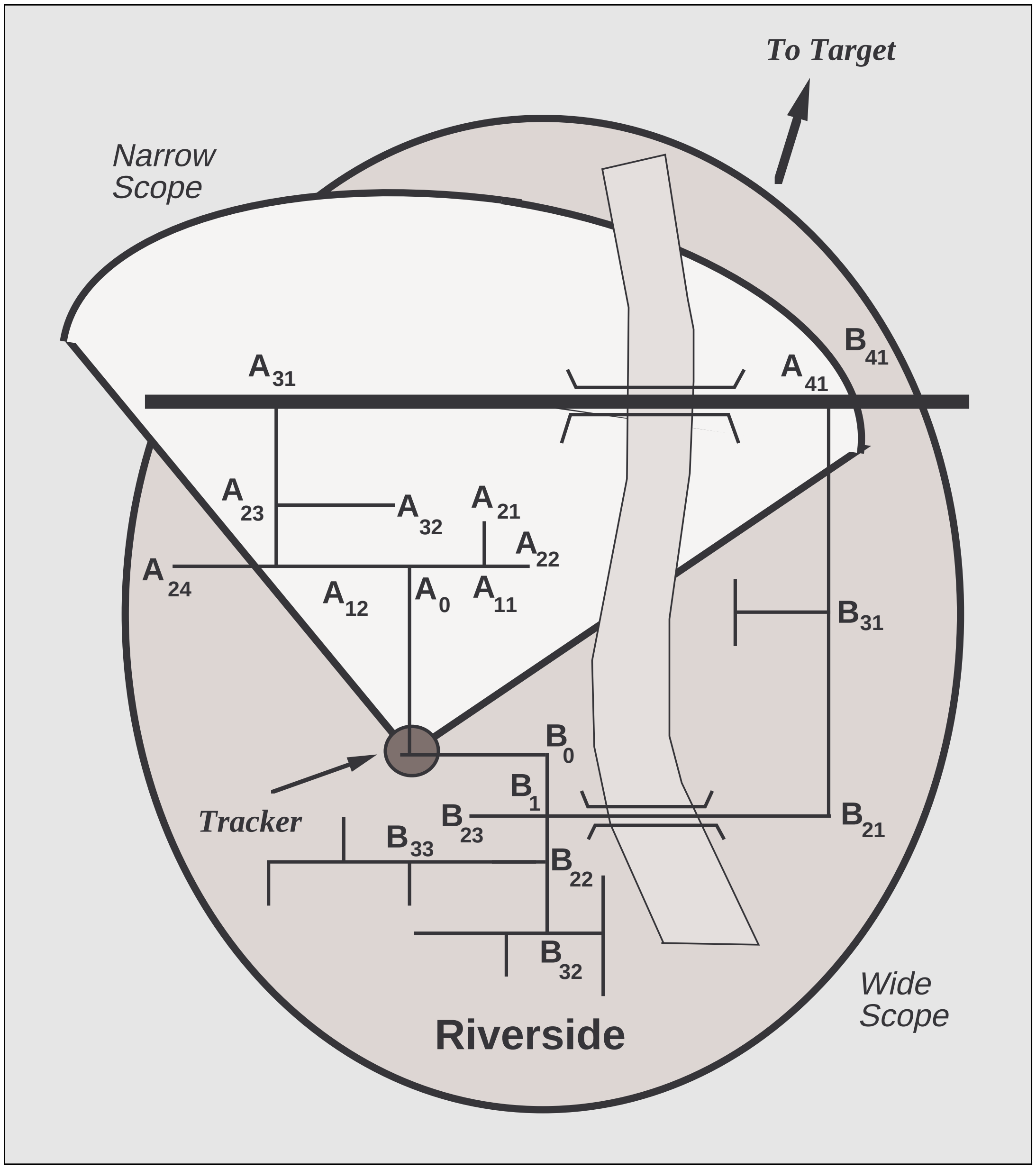


Figure 6. Scoping in the Application Example.

more information with each move, we can only look a few moves ahead within the time available. Both scopes bring additional knowledge and information to bear on the next move, and each approach can be the preferred approach based on circumstances.

If it were possible to plan the entire path from the start state to the goal state before starting execution, then scoping would not be necessary. Such an approach is possible in purely static environments. In our application example, this situation would correspond to the target vehicle being parked. The tracker could plan his/her entire route before he/she started.

Consider now the situation where the tracker cannot plan the entire route before starting and must plan the route while moving. For this case, the degree of scoping in time (intersections ahead) and space (azimuth of roads under consideration) should simply be maximized. The trade-off between time and space would be a function of the **regularity** of the environment. In our example, if the streets were perfectly rectilinear, then one would evaluate a relatively small azimuth and look as far forward in time as possible. Conversely, if the streets were highly irregular—corresponding to many dead-end streets, one-way streets, and other obstacles—then one would need to examine a wider azimuth to find the best route. To avoid backtracking, the level of scope should be such that it includes the irregularities that can affect the solution.

Dynamic problem solving (solving as you go) in static environments (parked target vehicle) allows one to trade storage for decreased execution time. In cases where one has to backtrack, the information about intersections evaluated previously but not taken can be stored. However, in dynamic environments corresponding to the target vehicle moving in an unpredictable fashion, the value of stored information about previously expanded nodes degrades with time. The appropriate level of scoping is now a function of both the regularity and the dynamics of the environment. In our car tracker example, it is not useful to plan a route too far ahead (in time) because the target will have changed position before you get there. In highly dynamic environments, one is forced to completely replan after each move. In the dynamic tracker example, this replanning corresponds to a new evaluation of the intersections within a given scope at every intersection, even though they might have been in your previous scoping window. The target has moved, and the relative value of following the path might have changed.

Some problem spaces can be structured naturally for increasing scope in time and space. Path-planning problems and scheduling problems fall in this category. For problem spaces that satisfy this property, the behavior of the actual solution quality is a function of whether an exact or a heuristic evaluation function (or error function) is available. If only a heuristic evaluation function is available, then one can at best guarantee that the expected solution quality, not the actual solution quality, will improve with time.

With scoping, we note the following: 1. The initial state and the goal state are not affected; what changes is the manner in which the goal state is approached. 2. In some cases, the problem space might have to be coerced to exhibit this property, which can potentially increase the worst-case or the average-case execution time of the task. The benefit of partial solutions along the way might or might not mitigate the increased response times. 3. This technique has the most impact on the selection of the next operator (move). In most real-time applications, it would be difficult to plan a static path from the current location of the tracker to the target. Because of unpredictable movement of the target and changing data, the path will have to be replanned after every move. In such cases, scoping has a large effect on limiting the time to select an operator for the next move. 4. Increasing the scope brings additional knowledge and information to bear on the choice of the next move, thereby increasing the probability that the right move is made. Increasing the probability of a correct move reduces the probability of backtracking and can potentially reduce the average-case execution time.

We also note that scoping is fundamentally different from approximating. In scoping, a given map is processed, but increasing information is brought to bear on the decision. In approximating, the map itself is modified. Hence, the resulting solution using approximating is necessarily less accurate, even if sufficient time was available to complete the search process.

Scoping is also different from pruning and ordering. Scoping determines the maximum fraction of the search space that will be searched. Thus, pruning and ordering are independently applied on the states that fall within a given scope.

Knowledge Retrieval

In the previous section, we considered the use of domain knowledge to reduce search variations at the problem-level search space or problem space. At each state in the problem space, knowledge was used to evaluate the operators in the current state and determine the next-best state. The process of extracting knowledge from the knowledge base and making it available to the search process at the problem-space level is called *knowledge retrieval.* It occurs at each state of the problem space, as shown in figure 7.

In analyzing the execution-time variations that are the result of search and backtracking at the problem-space level, we made the implicit assumption that the knowledge retrieval process at each problem-space state was bounded and took unit time. Unfortunately, in real-world systems, the knowledge retrieval time varies from one problem-space state to another and is difficult to characterize. Typical bounds for knowledge retrieval time tend to be exponential bounds, which are of no practical value in predicting the actual execution time (Forgy 1982).

We now consider the execution-time characteristics of the knowledge retrieval process, which occurs at each state of the problem

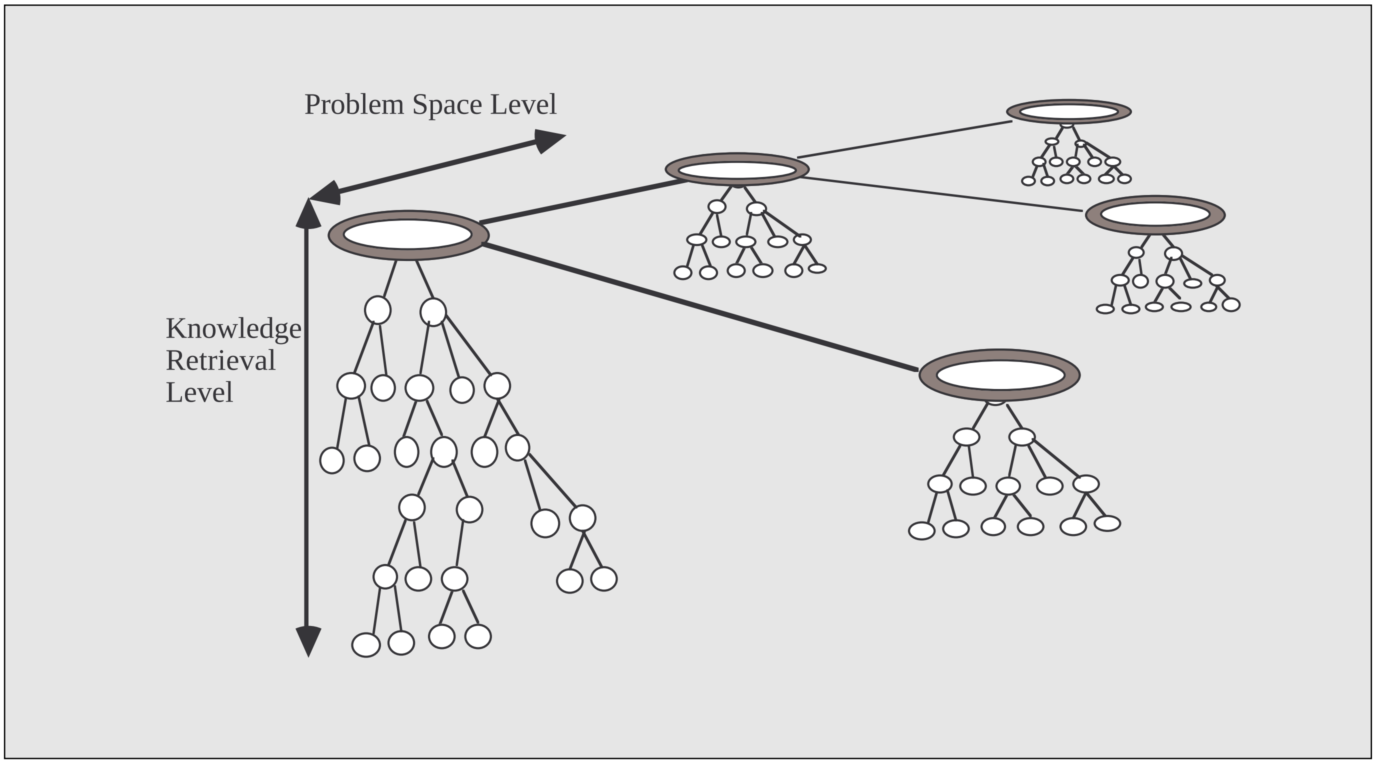


Figure 7. Search Spaces in Problem Solving.

space. To understand this issue and evaluate the options in reducing search variations at this level, let us first consider the nature of knowledge retrieval processing. We restrict our attention to immediate knowledge retrieval (that is, retrieving knowledge from the knowledge base or rule base only). All references to knowledge retrieval within the context of this article are limited to immediate knowledge retrieval only.

The primary function of the immediate knowledge retrieval process is to find all relevant domain knowledge that is applicable for evaluating a given state at the problem-space level. This process involves comparing the current state of the world (consisting of relevant observed data patterns) with the knowledge stored in the knowledge base (consisting of relevant and valid data patterns). This comparison requires pattern matching, which can be viewed as a search process. If domain knowledge had to be made available at run time to this (knowledge retrieval) search process, the process would require its own nested knowledge retrieval search, and so on, to infinity. To avoid an architecture that allows infinite nesting of knowledge retrieval search processes, we find that any domain knowledge that can be used to restrict knowledge retrieval search is best applied at design time. Thus, the nature of knowledge retrieval search and the availability of domain knowledge to control the knowledge retrieval search are different from the availability of domain knowledge to control the search at the problem-space level.

Observation: The problem-level search space allows the use of both dynamic and static domain knowledge in reducing problem-solving variations, whereas the immediate knowledge retrieval process in nonlearning systems allows the use of only static domain knowledge. Learning systems have the potential to acquire knowledge during run time and apply it to reduce execution-time variations of the knowledge retrieval process, but this research challenge remains open.

In the previous section, we hypothesized that the only way to manage search variations is to use domain knowledge for pruning, ordering, approximating, and scoping. Because using run-time knowledge is difficult, a viable approach to using domain knowledge is to partition the data and the knowledge base so that relevant data can be matched with the relevant partitions. The implications for managing the execution-time variations at the knowledge retrieval level are examined in the following subsections.

Recent research in real-time production systems evaluates statistical approaches to predict the run time of the knowledge retrieval process (Barachini, Mistelberger, and Gupta 1992).

Knowledge Retrieval Pruning

The amount of processing required in the knowledge retrieval phase depends on the amount of knowledge in the system, the size of the state (data), and the amount of data to which each piece of knowledge is potentially applicable. In many systems, the process of knowledge retrieval is essentially a pattern-matching process: finding valid patterns in data elements as a function of the patterns stored in the knowledge base (Acharya and Kalb 1989; Nayak, Gupta, and Rosenbloom 1988; Scales 1986; Forgy 1984). Thus, the pattern-match time is a function of the number of data elements, the set of possible relations, and the number of elements in the knowledge base. To reduce the total work at this level, there are only three degrees of freedom: (1) reduce the relations in the knowledge base (using knowledge partitioning), (2) reduce the data elements (using data partitioning), and (3) reduce the relations that are possible in the first place by restricting the language (Tambe and Rosenbloom 1989) (restricting expressiveness).

Doing the partitioning ensures that only the relevant data are matched with the relevant knowledge base. One of the primary causes of large execution-time variations in knowledge retrieval, as well as one of the biggest sources of inefficiency, is unnecessary match processing that is later discarded (Tambe and Rosenbloom 1989). The unnecessary match processing is because the knowledge retrieval inference engine retrieves all possible instantiations for a particular rule and then selects one by conflict resolution. The rest are not used in the current inference cycle. The search to find the instantiations is essentially a blind search and has limited aid from domain knowledge. Restricting the expressiveness of the language or moving the search from the knowledge retrieval space to the problem-space level, in addition to partitioning, would limit unwanted search at the knowledge retrieval level.

As a guiding principle, by applying a relevance filter to both the data and the knowledge base, we hope to limit the match processing to only useful processing, resulting in the following options:

Partitioning the knowledge space: By partitioning the knowledge space, we are able to avoid searching sections of the knowledge space that contain knowledge that is a priori known to not be applicable to particular pieces of data. In the tracking example, the knowledge base contains knowledge about intersections, road segments, and their connectivity. When processing a subproblem at the level of a neighborhood, it is not necessary to consider intersections and road segments in other neighborhoods. Thus, partitioning the knowledge base by neighborhood can restrict the knowledge retrieval time. Also, although it is easy to talk about partitioning a fact base (for example, partitioning streets by neighborhoods), it is more difficult to partition higher forms of knowledge (for example, active relationships and rules of inference) that could have application across multiple partitions.

Partitioning the data: Many pieces of knowledge express relations, desired or otherwise, between multiple pieces of data. Partitioning the data allows us to avoid considering sets of data that are a priori known to not belong to the relation. In the tracking example, continuous traffic information for all road segments is flowing into the system. Processing all this information all the time can be expensive computationally and unnecessary. (Besides, the data are getting stale anyway, so why process them if they are not currently required?) Partitioning these data by neighborhood and processing the data for the current neighborhood would restrict the knowledge-processing time.

Restricting expressiveness: Highly expressive knowledge representations allow a large number of relations to be represented. When a part of the state changes, a large number of potential relations with the rest of the state have to be checked. Representation formalisms that restrict expressiveness a priori restrict the number of relations that need to be checked during every state change. In the tracking example, we could use a representation formalism that allows unstructured sets to be represented in working memory. However, if we use structured sets (for example, intersection I1 is north of intersection I3 as opposed to intersection I1, which is adjacent to intersection I3), the number of potential relations that have to be checked at run time is limited, allowing polynomial bounds on knowledge retrieval time (Tambe and Rosenbloom 1989).

Knowledge Retrieval Ordering

Run-time ordering of states based on domain-specific heuristic evaluation functions is not possible at the knowledge retrieval level. At best, some knowledge-lean, domain-independent heuristics can be applied. Hence, the options for ordering at this level are more restricted than at the problem-level search space.

However, even though domain knowledge cannot be used to order the processing at run time, an implicit ordering in the evaluation can be built into the match process (because of the serialization of processing). A programmer writing application code can use this knowledge, along with knowledge of the application domain, to tailor the application code to reduce knowledge retrieval search at run time.

Knowledge Retrieval Approximation

Approximation is another technique that works at the problem-space level. In dealing with approximation at the knowledge retrieval level, we first distinguish between knowledge of an approximation and an approximation of knowledge. *Knowledge of an approximation* is knowledge to deal with data clusters and knowledge to deal with data approximations. *Approximation of knowledge* is the deliberate ignoring of more specific or corroboratory knowledge because of limitations in time.

Providing support at the knowledge retrieval level to different forms of approximation is equivalent to providing support for different rule-set partitions because the knowledge retrieval level cannot distinguish between knowledge of approximations and other forms of knowledge.

However, to support approximation of knowledge at run time, the knowledge retrieval process must have the ability to determine (at run time) the level of approximation that is appropriate for a given situation. Given the nature of the knowledge retrieval process, approximation will require domain knowledge to be hard coded into the knowledge retrieval search as well as make the knowledge retrieval process arbitrarily complex. Also, an approximation of knowledge could be counterproductive because the knowledge that is ignored could potentially cause unnecessary states to be visited at the problem-space level (each requiring a whole knowledge retrieval step) as opposed to the current step and the incremental knowledge retrieval time that would be required to complete processing.

We feel that exploiting notions of time and approximation at the knowledge retrieval level is hard and of questionable value. However, these notions can effectively be exploited at the problem-space level. Structuring the process to move the search from the knowledge-space level to the problem-space level is the solution to this problem and alleviates the need for knowledge approximations.

Knowledge Retrieval Scoping

There is no notion of scoping at the knowledge retrieval level. Any attempt to implement scoping results in arbitrary restrictions on the match process, resulting in potential correctness and completeness problems.

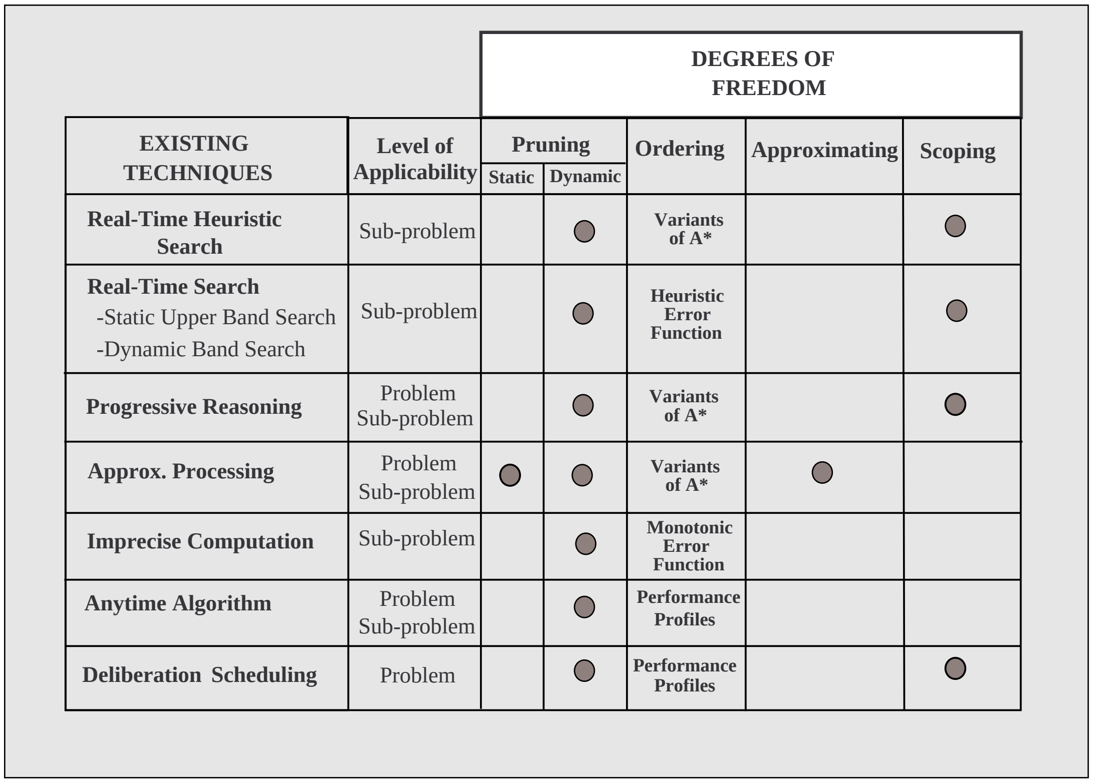
Thus, domain knowledge can be used at design time to prune, order, and structure the knowledge retrieval space. We now examine how existing approaches map to our structured approach. The existing real-time problem-solving strategies are mapped to the degrees of freedom that they exploit.

Mapping Existing Techniques

Over the past few years, various techniques have been developed by the AI and real-time communities for generating satisficing solutions in limited time for search and nonsearch problems. These techniques include imprecise computation (Chung, Liu, and Lin 1990), real-time search (Korf 1990), anytime algorithms (Dean and Boddy 1988), and approximate processing (Lesser, Pavlin, and Durfee 1988). Our structured view is an attempt to map the nature of the entire design space. Each technique represents a different point in the design space of real-time AI systems. Our attempt to map these techniques is made difficult by the terminology differences between the real-time community and the AI community and the lack of precise definitions in some cases. In this section, we examine these techniques in greater detail and attempt to map them to the structured view of real-time problem solving. Figure 8 summarizes the techniques and the degrees of freedom they exploit.

Real-Time Search Algorithms

A number of algorithms have been developed to address searches in real time. In the real-time heuristic search techniques proposed by Korf (1990), a time constraint is associated with node expansion, which determines the amount of lookahead in making a decision for the node. These techniques rely on the least-commitment strategy, allowing them to be applicable equally in static, as well as dynamic, environments (Ishida and Korf 1991). Once all the nodes within the search



		DEGREES OF FREEDOM				
EXISTING TECHNIQUES	Level of Applicability	Pruning		Ordering	Approximating	Scoping
		Static	Dynamic			
Real-Time Heuristic Search	Sub-problem		●	Variants of A*		●
Real-Time Search -Static Upper Band Search -Dynamic Band Search	Sub-problem		●	Heuristic Error Function		●
Progressive Reasoning	Problem Sub-problem		●	Variants of A*		●
Approx. Processing	Problem Sub-problem	●	●	Variants of A*	●	
Imprecise Computation	Sub-problem		●	Monotonic Error Function		
Anytime Algorithm	Problem Sub-problem		●	Performance Profiles		
Deliberation Scheduling	Problem		●	Performance Profiles		●

Figure 8. Existing Techniques and the Degrees of Freedom They Exploit.

horizon are evaluated, a single move is made in the direction of the best next state; the entire process is repeated under the assumption that after committing the first move, additional information from an expanded search frontier can result in a different choice for the second move than was indicated by the first search. These techniques map to search within a subproblem at a single level of abstraction, using scoping in time (range) at each state to perform the lookahead. Scoping in space (azimuth) is kept at a maximum. Because these searches have to commit to a move before the entire solution is found, optimality of the overall solution-execution sequence is not guaranteed. The search techniques for optimization in limited time proposed by Chu and Wah (1991) look for the solution with the best ascertained approximation degree. The search space is reduced using an approximation function to eliminate nodes that deviate beyond a threshold from the current best solution. Their approach can be viewed as a way of implementing scoping in our framework. These search techniques are applied within a subproblem at a single level of abstraction. The approximations to the evaluation function change the scope of the search and do not affect the nature of the search space. Hence, they do not count as approximations in our structured view. Because the approximations are used to perform pruning, the optimal solution can be pruned away. The goal state is compromised because the optimal solution is not always found, with the assumption that the nonoptimal solution that is found is acceptable. Also, by its very nature, these techniques are applicable to problems that are static as the search proceeds. Dynamic situations require restarting the search process with a new set of data.

Other techniques, such as static upper band search and dynamic band search proposed by Chu and Wah (1992), restrict the breadth and depth of the search and are modified forms of beam search. In our framework, these techniques are applicable within a subproblem at a single level of abstraction. Even though these techniques restrict the breadth and depth of the search (by limiting the nodes in the active list), the decision at each node is made only after considering the immediate successor nodes. Hence, these techniques do not implement the type of scoping we discuss in our framework (which is a lookahead to a certain breadth and depth for each move). These techniques are only applicable in domains that are static. If the domain is dynamic (for example, target moves), the searches have to be restarted with a new set of data.

Progressive Reasoning

Progressive reasoning is a technique to analyze the situation to depth 1, then depth 2, then depth 3, and so on, until the deadline is reached (Winston 1984). Another variant of this approach is *depth-first iterative deepening* (Korf 1985), in which the search is restarted from the root node with an incremental depth bound after the current depth bound is reached. Within a given subproblem, these techniques can be mapped to scoping in time, with maximum scoping in space (that is, the azimuth is at a maximum; only the range is increased with additional time). Progressive reasoning has also been used to include incremental detail as the search progresses. In our structure, progressive reasoning maps to using multiple subproblems, each at a different level of abstraction, and structuring the search process to progressively traverse the abstraction levels. In our tracking example, moving between subproblem 2 (at the level of highways) to subproblem 3 (in the neighborhood of the tracker) would be an instance of this approach.

Approximate Processing

In its broad sense, *approximate processing* is a collection of techniques for dealing with inference under uncertainty, in which the underlying logic is approximate or probabilistic rather than exact or deterministic (Zadeh 1985). Approximate processing is relevant in situations in which there is not enough time to find the optimal solution. A number of techniques (Lesser, Pavlin, and Durfee 1988; Decker, Lesser, and Whitehair 1990) have been developed in this area to make tradeoffs in solution quality along three axes: (1) completeness, (2) precision, and (3) certainty. These techniques compromise the accuracy with which the goal state is achieved.

This approach exploits the approximation degree of freedom described in the structured view. At the subproblem level, approximation in precision can be mapped to value approximation and the reducing of the cardinality of a given characteristic. Approximation in completeness can be mapped to ignoring some of the characteristics on which the solution is dependent. Approximation in certainty can be mapped to a combination of the simplification of the functional relationship between the characteristics and the ignoring of some of the characteristics on which the solution is based.

At the problem level, the approximate processing approach relies on making method approximations. Method approximation relies primarily on function simplification.

The mapping of approximate processing is illustrated using the tracking example. Figure 9 shows how the tracking example can be made to conform to the model of approximate processing (Garvey and Lesser 1992). We note the similarities between subproblem 2 in figure 9 and in figure 4. Approximation within a subproblem is illustrated in the different problem spaces generated at the levels of approximation of highways only; highways and major roads; and, the most detailed version, all roads. Method approximation is illustrated by the presence of the arc bypassing the solution of subproblem 2 using a default solution instead. (The default solution might have been generated in the previous pass using older data). In the example illustrated, the only viable reason for skipping subproblem 2 entirely is when the target is in the same neighborhood as the tracker. Skipping an entire subproblem is possible in situations in which the input and the output of the skipped subproblem have the same form.

Imprecise Computation

Imprecise computation relies on making results that are of poorer but acceptable quality available on a timely basis when results of the desired quality cannot be produced in time. This approach, as formally defined in Chung, Liu, and Lin (1990), is applicable to those processes that are designed to be monotone (a process is called a *monotone process* if the accuracy of its intermediate result is nondecreasing as more time is spent to produce the result). Thus, imprecise computation exploits well-defined error functions that guarantee monotonicity.

As such, the imprecise computation approach maps well to iterative processes. Even though this technique originally had no relation to AI, of late, the interpretation of this technique is being extended to address search-based processes. Note that error-function formalities originally associated with imprecise computation are sacrificed because monotonicity cannot be guaranteed in search-based domains.

Anytime Algorithms

Anytime algorithms are algorithms that return some answer for any allocation of computation time and are expected to return better answers when given more time (Dean and Boddy 1988). This approach requires an ini-

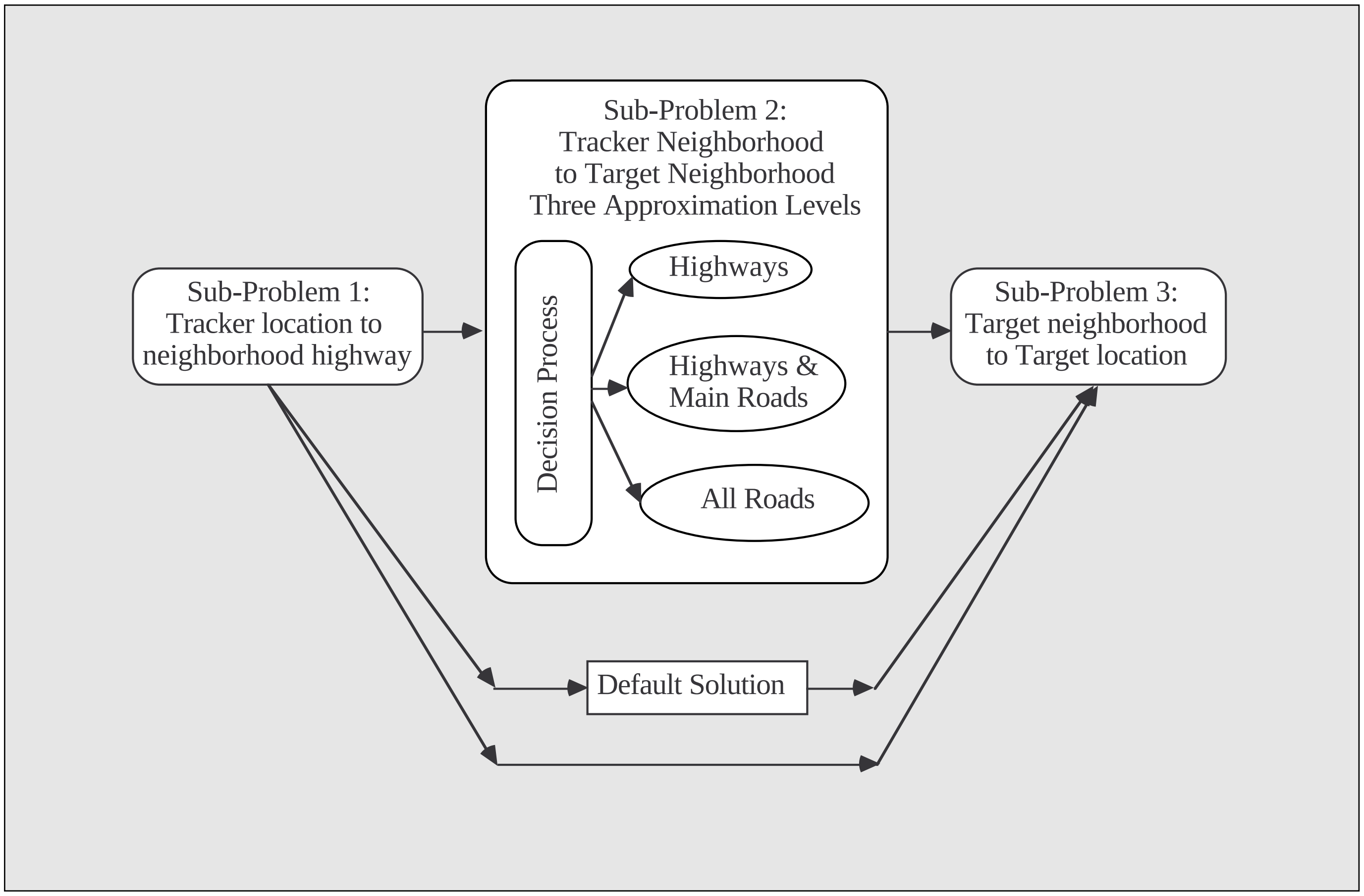


Figure 9. Mapping Approximate Processing to the Tracking Example.

tial solution whose cost is known and some measure of how this cost is expected to improve with deliberation time. The primary contribution of the anytime algorithm approach is the introduction of time as an additional degree of freedom in allocating resources for computation.

Because of its rather broad definition, anytime algorithms appear more to be an attempt to define an anytime property for real-time problem-solving systems than a specific technique. As such, the techniques published under the anytime umbrella march across the full spectrum and exploit various combinations of the fundamental degrees of freedom. It would be possible to map specific implementations of anytime algorithms to the structured view. The robot path-planning example described in Boddy and Dean (1989) breaks the problem into subproblems that exploit scoping in time and space. Deliberation scheduling was used to implement decision making. Major contributions of this work include the specification of deliberation scheduling as a sequential decision-making problem and the use of expectations in the form of performance profiles to guide search.

Over the last few years, a lot of work has gone into defining interesting special cases of performance profiles (for example, Zilberstein and Horvitz). One of the difficulties in implementing anytime algorithms is generating performance profiles that are an integral part of most anytime algorithms. In search-based domains, performance profiles are difficult to generate because of the large variance in the execution time of the search process. The large variance results in a wide spread between the upper and lower bounds of the performance profile, resulting in low predictability. Conditional performance profiles help mitigate some of the problems in circumstances where the initial conditions do not allow for accurate low-variance performance predictions.

In nonsearch domains with well-defined error functions, more precise performance profiles are possible because of lower execution-time variations. In these domains, this approach is similar to the imprecise computa-

Search-based problems fall into two broad classes: iterative search algorithms and generalized search processes.

Iterative search algorithms

Iterative search algorithms (search without backtracking), such as the binary search of a dictionary and the Newton-Raphson method, are search techniques in domains where a perfect evaluation function (or an exact error function) is available. At each step, the evaluation function guarantees taking you closer along the path to the goal state. There is no backtracking—only iterative convergence. Hence, it is possible to guarantee the monotonicity of the actual solution quality. Execution-time variations are solely the result of data dependencies: start state and goal state. Bounds on the worst-case execution time typically grow logarithmically as a function of the size of the problem space. The imprecise computation approach (Chung, Liu, and Lin 1990) developed in the real-time community falls into this category.

Generalized search processes

Generalized search processes (search with backtracking), such as heuristic search, are search techniques in domains where only an inexact, heuristic evaluation function (or error function) is available. At each step, the heuristic evaluation function takes you closer to the goal state in an expected sense. However, it is possible to be wrong, requiring backtracking. Therefore, it is not possible to guarantee the monotonicity of the actual solution quality. At best, heuristic search functions can only improve the expected solution quality in a monotonic fashion. Execution-time variations are a function of both data dependencies and backtracking. Bounds on the worst-case execution time typically grow exponentially as a function of the size of the problem space. The deliberation scheduling technique (Boddy and Dean 1989) developed in the AI community assumes a class of underlying anytime algorithms that provide monotonically improving solution quality. Although the anytime algorithm approach can monotonically improve the expected solution quality, the actual solution quality cannot be guaranteed to improve monotonically because of backtracking.

tion approach without the mandatory computation assumption. In real life, it is not clear to what extent this difference is a defining one. Most applications require some computation time before even a feasible default solution can be generated. In general, the default solution at time zero could be to ensure safety; for example, the robot could stop moving or a plane could continue flying in the same direction, although in both cases, it would be difficult to ensure that the default solution indeed guarantees safety.

Thus, all techniques, whatever their label, attempt to find satisficing solutions in limited time. We mapped real-time search techniques to be applicable within a single level of abstraction. Approximate processing techniques span multiple levels of abstraction. The distinction between imprecise computation and anytime algorithms continues to blur as the interpretation of imprecise computation continues to broaden. The current trend seems to distinguish the two based on the presence or absence of a mandatory computation-time requirement, but we believe that this difference is purely cosmetic. We find that existing techniques exploit different combinations of the fundamental degrees of freedom in generating satisficing solutions in limited time.

Summary and Limitations

The key challenge in real-time problem solving is constraining the execution time of naturally exponential search problems in a best-so-far fashion such that satisficing solutions can be guaranteed in time without severe underuse of system resources (Paul et al. 1991). This article examined the execution-time variations of real-time problem solving within the context of the Newell-Simon problem-space hypothesis. We conjectured that there are at least four degrees of freedom in reducing search variations and structuring the search space and the search process to produce best-so-far solutions. We considered the two levels of problem-solving search: (1) the problem-level search space and (2) the knowledge retrieval level. The problem-level search space was divided into smaller search spaces called subproblems. Each subproblem was defined to be at a single level of abstraction. The effect of each of these degrees of freedom on the execution time of the task was discussed using a tracking example as a conceptual aid.

In evaluating execution-time effects, we find that pruning in its many manifestations is the only technique that reduces the worst-case execution time without compromising the goal state. Approximate processing can reduce the worst case but compromises the goal state. Other techniques, such as scoping, reduce the average-case execution time and provide a best-so-far solution property. We also examined the application of these degrees of freedom at the knowledge retrieval level. We find that partitioning is the most effective technique at this level because of limitations in using domain knowledge.

We then examined existing approaches to real-time problem solving and mapped them to our structured approach based on the fundamental degrees of freedom. Within this context, the different techniques were shown to exploit compositions of these fundamental degrees of freedom.

Acknowledgments

We thank Milind Tambe, Sarosh Talukdar, Herbert Simon, Dorothy Setliff, Thomas Dean, Jane Liu, Alan Garvey, Richard Korf, Marcel Schoppers, and the anonymous reviewers for their suggestions to improve the clarity and the content of this article.

References

Acharya, A., and Kalp, D. 1989. Release Notes on ParaOPS5 4.4 and CParaOPS5 5.4. School of Computer Science, Carnegie Mellon Univ.

Barachini, F.; Mistelberger, H.; and Gupta, A. 1992. Run-Time Prediction for Production Systems. In Proceedings of the Tenth National Conference on Artificial Intelligence, 478–485. Menlo Park, Calif.: American Association for Artificial Intelligence.

Boddy, M., and Dean, T. 1989. Solving Time-Dependent Planning Problems. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 979–984. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Chapman, R. L.; Kennedy, J. L.; Newell, A.; and Biel, W. C. 1959. The System's Research Laboratory's Air Defense Experiments. *Management Science* 5:250–269.

Chu, L., and Wah, B. 1992. Solution of Constrained Optimization Problems in Limited Time. In Proceedings of the IEEE Workshop on Imprecise and Approximate Computation, 40–44. Washington, D.C.: IEEE Computer Society.

Chu, L., and Wah, B. 1991. Optimization in Real Time. In Proceedings of the Twelfth Real-Time Systems Symposium, 150–159. Washington, D.C.: IEEE Computer Society.

Chung, J.; Liu, J.; and Lin, K. 1990. Scheduling Periodic Jobs That Allow Imprecise Results. *IEEE Transactions on Computers* 39(9): 1156–1174.

Dean, T., and Boddy, M. 1988. An Analysis of Time-Dependent Planning. In Proceedings of the Seventh National Conference on Artificial Intelligence, 49–54. Menlo Park, Calif.: American Association for Artificial Intelligence.

Decker, K. S.; Lesser, V. R.; and Whitehair, R. C. 1990. Extending a Blackboard Architecture for Approximate Processing. *Journal of Real-Time Systems* 2(1–2): 47–79.

Dodhiawala, R.; Sridharan, N. S.; Raulefs, P.; and Pickering, C. 1989. Real-Time AI Systems: A Definition and an Architecture. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 256–264. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Forgy, C. L. 1984. The OPS83 Report. Technical Report, CMU-CS-84-133, Computer Science Dept., Carnegie Mellon Univ.

Forgy, C. L. 1982. RETE: A Fast Algorithm for the Many Pattern–Many Object Pattern Match Problem. *Artificial Intelligence* 19(1): 17–37.

Garvey, A., and Lesser, V. 1992. Scheduling Satisficing Tasks with a Focus on Design-to-Time Scheduling. In Proceedings of the IEEE Workshop on Imprecise and Approximate Computation, 25–29. Washington, D.C.: IEEE Computer Society.

Ishida, T., and Korf, R. E. 1991. Moving Target Search. In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, 204–210. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Jeffay, K.; Stanat, D.; and Martel, C. 1991. On Non-Preemptive Scheduling of Periodic and Sporadic Resources. In Proceedings of the IEEE Real-Time Systems Symposium, 129–139. Washington, D.C.: IEEE Computer Society.

Knoblock, C. A. 1990. Learning Abstraction Hierarchies for Problem Solving. In Proceedings of the Eighth National Conference on Artificial Intelligence, 923–928. Menlo Park, Calif.: American Association for Artificial Intelligence.

Korf, R. E. 1990. Real-Time Heuristic Search. *Artificial Intelligence* 42(2–3): 189–211.

Korf, R. E. 1987. Planning as Search: A Quantitative Approach. *Artificial Intelligence* 33(1): 65–88.

Korf, R. E. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence* 27(1): 97–109.

Laffey, T. J.; Cox, P. A.; Schmidt, J. L.; Kao, S. M.; and Read, J. Y. 1988. Real-Time Knowledge-Based Systems. *AI Magazine* 9(1): 27–45.

Lesser, V. R.; Pavlin, J.; and Durfee, E. 1988. Approximate Processing in Real-Time Problem Solving. *AI Magazine* 9(1): 49–62.

Liu, C. L., and Layland, J. W. 1973. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM* 20(1): 46–61.

Nayak, A.; Gupta, P.; and Rosenbloom, P. 1988. Comparison of the RETE and TREAT Production Matchers for SOAR (a Summary). In Proceedings of the Seventh National Conference on Artificial Intelligence, 693–698. Menlo Park, Calif.: American Association for Artificial Intelligence.

Newell, A., and Simon, H. 1972. *Human Problem Solving*. Englewood Cliffs, N.J.: Prentice-Hall.

Paul, C. J. 1993. A Structured Approach to Real-Time Problem Solving. Ph.D. thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon Univ.

Paul, C. J.; Acharya, A.; Black, B.; and Strosnider, J. K. 1991. Reducing Problem-Solving Variance to Improve Predictability. *Communications of the ACM* 34(8): 64–93.

Paul, C. J.; Holloway L. E.; Strosnider, J. K.; and Krogh, B. H. 1992. An Intelligent Reactive Scheduling and Monitoring System. *IEEE Control Systems* 12(3): 78–86.

Scales, D. J. 1986. Efficient Matching Algorithms for the SOAR/OPS5 Production System. Technical Report, KSL-86-47, Knowledge Systems Lab., Stanford Univ.

Schwan, K., and Zhou, H. 1992. Dynamic Scheduling of Real-Time Tasks and Real-Time Threads. *IEEE Transactions on Software Engineering* 18:736–748.

Sha, L., and Goodenough, J. B. 1990. Real-Time Scheduling Theory and ADA. *IEEE Computer* 23:53–62.

Sprunt, B.; Sha, L.; and Lehoczky, J. 1989. A Periodic Task Scheduling for Hard Real-Time Systems. *Journal of Real-Time Systems* 1(1): 27–60.

Stankovic, J. A. 1988. Misconceptions about Real-Time Computing: A Serious Problem for the Next-Generation Systems. *IEEE Computer* 21:10–19.

Tambe, M., and Rosenbloom, P. 1989. Eliminating Expensive Chunks by Restricting Expressiveness. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 731–737. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

VanTilborg, A., and Koob, G. M. 1991. *Foundations of Real-Time Computing, Scheduling, and Resource Management*. New York: Kluwer Academic.

Wensley, J.; Lamport, L.; Goldberg, J.; Green, M. W.; Levitt, K.; Melliar-Smith, P.; Shostak, R.; and Weinstock, C. 1978. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE* 66(10): 1240–1255.

Winston, P. H. 1984. *Artificial Intelligence*. 2d ed. Reading, Mass.: Addison-Wesley.

Zadeh, L. A. 1985. Foreword. In *Approximate Reasoning in Expert Systems*. New York: North-Holland.

Jay K. Strosnider is an associate professor in the electrical and computer engineering department at Carnegie Mellon University. He has ten years industrial experience developing distributed real-time systems for submariens and six years academic experience trying to formalize real-time systems engineering. His current research focus is on intergrating wide-ranging technologies within a scheduling theoretic framework.

C. J. Paul is a member of the staff at IBM Austin. His research interests include computer architecture, operating systems, and real-time artificial intelligence.