Approximation Algorithms for Planning and Control

Mark Boddy and Thomas Dean[1]
Department of Computer Science
Brown University
Box 1910, Providence, RI 02912

Abstract

A control system operating in a complex environment will encounter a variety of different situations, with varying amounts of time available to respond to critical events. Ideally, such a control system will do the best possible with the time available. In other words, its responses should approximate those that would result from having unlimited time for computation, where the degree of the approximation depends on the amount of time it actually has. There exist approximation algorithms for a wide variety of problems. Unfortunately, the solution to any reasonably complex control problem will require solving several computationally intensive problems. Algorithms for successive approximation are a subclass of the class of *anytime algorithms*, algorithms that return answers for any amount of computation time, where the answers improve as more time is allotted. In this paper, we describe an architecture for allocating computation time to a set of anytime algorithms, based on expectations regarding the value of the answers they return. The architecture we describe is quite general, producing optimal schedules for a set of algorithms under widely varying conditions.

1 Introduction

In the best of all worlds, there are infinite computing resources. Unfortunately, this is not the best of all worlds, and, while computing resources are steadily becoming cheaper, there are problems that occur routinely in robotics and process planning that will exhaust any resources that we might plausibly bring to bear. We refer to the class of *NP-hard* problems that, so far, have eluded the best efforts of algorithm designers to provide efficient solutions, and will likely continue eluding them.

Of course, the *NP-hard* problems are not the only obstacle to designing effective control algorithms. There are plenty of problems (*e.g.*, various shortest-path problems) for which there exist polynomial-time solutions that run too slowly on existing machines to support real-time control. In some cases, we can compensate by caching results in tables and computing the answers to problems in real time by table lookup. This approach has its own drawbacks, however, as tables require storage and for many problems the required storage is more than is practical. In addition, as our notion of control expands to encompass more and more complicated sorts of behavior, the number of functions that we would have to tabulate becomes quite large, making the idea impractical.

One conclusion to be drawn from the above is that for some problems we cannot expect the best possible answers; if we want to tackle certain problems, we will have to satisfy ourselves with approximate solutions. Computer science in general and artificial intelligence in particular has been concerned for some time with approximate solutions, and as a consequence many algorithms exist for well-known problems. We can't, however, apply such algorithms directly since these well-known problems are just subproblems of the complex sort of control problems encountered in robotics and process planning. What is needed is a method for integrating solutions to these simpler well-known problems so as to provide reasonable performance for the more complex problems.

In this paper, we present an approach to dealing with problems in real-time planning and control. Our approach involves using a particular sort of algorithm called an *anytime* algorithm. An anytime algorithm can be interrupted at any point during its execution to return an answer whose utility or expected value

[1]This work was supported in part by the National Science Foundation under grant IRI-8612644 and by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Air Force Office of Scientific Research under Contract No. F49620-88-C-0132.

is a monotonic increasing function of the time spent computing. The more time available the better the answer returned. A set of such algorithms can be orchestrated to provide solutions to various sorts of control problems that are in some well-defined sense optimal. Our techniques are particularly suited to applications in which the response time for certain critical events is subject to wide variation, and applications that require the solution to several independent subproblems each of which is compute intensive. Such applications are referred to as *time dependent*. We begin our discussion with an introduction to the class of anytime algorithms.

2 Anytime Algorithms

Almost any algorithm can be trivially turned into an anytime algorithm by embedding it in a second algorithm that runs the original algorithm as an inferior process. At any point when the parent process is interrupted and asked for an answer, it checks to see if the inferior process has terminated; if so it returns the answer generated by the inferior process, and otherwise it returns some default answer. The utility of the answers returned by the parent process is a trivial monotonic increasing function of the time spent computing: a step function with a single step. In most cases, however, we can provide a more useful anytime algorithm (*i.e.*, one which produces a succession of increasingly useful results). For instance, many search algorithms employ some sort of a metric for determining if one answer is better than another. At all times, the algorithm keeps track of the best answer computed so far. Such an algorithm could easily be designed to return its current best answer at any point in the computation.

For certain problems in the complexity class *NP*, while there are no known efficient algorithms that compute the exact answers in polynomial time, there exist approximation algorithms that can be shown empirically to provide good answers in a small number of steps. Rather than use complicated methods for choosing the best of some possibly exponential number of alternatives to explore, these algorithms flip coins to determine where to search next. A good example of this sort of algorithm is a probabilistic algorithm for testing primality [Harel, 1987]. This algorithm makes use of the fact that with probability approximately $\frac{1}{2}$, any of the numbers less than the number being tested can serve as a *witness* to its being composite. Finding a witness establishes that the number is not prime. That a number chosen at random is not a witness increases the probability that the number being tested is prime. The time necessary to run this algorithm depends on the probability bound required; the more points tested, the smaller the probability that we will falsely identify a number as a prime. An anytime algorithm for primality testing using this approach would continue choosing numbers at random and testing them as witnesses until it was interrupted (or determined that the number was composite), and then return the probability that the number was in fact a prime.

Another approach to combinatoric problems is to use approximation algorithms which search a smaller space (they are "approximate" because the optimal answer may not be in the reduced solution space). An example of this type of algorithm is the 2-OPT algorithm used for generating approximations to instances of the traveling salesman problem (TSP). 2-OPT begins with a cheaply generated tour that includes each city specified in the TSP instance. It then chooses two arcs in the tour, removes them, and reconnects the disconnected cities to form a new tour of smaller cost. In the standard approach, this cycle is repeated until there is no pair which can be exchanged to improve the tour. It has been shown empirically that running 2-OPT to completion produces tours which average within about 8% of the cost of the optimal tour. There are more complicated edge-exchange algorithms that do better [Lin and Kernighan, 1973]. An anytime algorithm implemented using 2-OPT will exchange pairs of arcs until it is interrupted and asked for an answer, at which point it returns the current tour.

In any interesting control problem, there are lots of different things that must be computed. We may have anytime algorithms for each individual problem, but what we need is some way of coordinating their behavior to produce a composite solution that makes optimal use of the available processor time. In order to engineer such coordination, we need two things: reasonably accurate expectations regarding the utility of the results returned by anytime algorithms as a function of computation time, and some strategy for using these expectations to allocate processor time. The first is relatively easy if we have the luxury of testing our algorithms on real or simulated data; we simply run the anytime algorithms repeatedly and gather statistics on the accuracy of the results obtained as a function of computation time. The second requirement can be more difficult to satisfy, and we devote the following sections to its discussion.

3 Scheduling Anytime Algorithms

The processes that we seek to control generally cannot be halted to wait for the controller to compute a response. However, we often have some idea of how much time is available for computing a re-

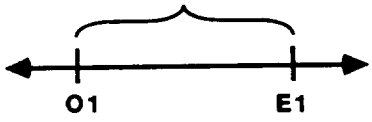

Figure 1: Predicting critical events

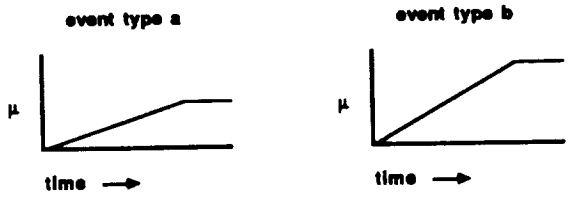

Figure 2: Performance profiles

Figure 3: Deliberation scheduling

sponse. There are a significant number of control problems that can be viewed in terms of reacting to predicted events, employing some model to predict critical events and computing functions to determine how best to respond to those critical events. Figure 1 depicts a time-line showing an observation O1 which can be used to predict the occurrence of a critical event E1. In this simple example, the time between the observation and the predicted occurrence of the event is the time available to compute a response. In tracking a ping-pong ball, for instance, one can predict the time until impact and, hence, the time available to think about how to orient the paddle and take whatever steps are required move it into that orientation. In the traditional approach to control, a discrete control algorithm samples the data at regular intervals, computes a control action, and then executes that action. The control algorithm has a fixed response time. If the sampling interval changes, then the algorithm has to be changed. In many control problems encountered in robotics, sample rates will depend on how quickly a robot can position a sensor, take a reading, and interpret the results. Ideally, the sampling interval will not matter; the controller will do the best it can with the time available.

The robot control problem is complicated by the fact that there may be more than one process to be controlled at the same time. Many problems in control involve coordinating multiple processes. In guiding a mobile robot, the process of avoiding obstacles has to be coordinated with the process of navigating through doorways. Some processes must be monitored and adjusted frequently. In other cases, such as coordinating an assembly process with a parts inventory control process, there is more time between critical events but the parameter adjustments also take more time. Given the problem of coordinating the process of planning a route with the process of driving a car, the two processes have very different utilities; taking a little more time to get there is worth avoiding an accident. Resources such as processor time and access to sensors will need to be allocated to competing controllers. This should happen in a principled way, *i.e.*, so that the resources available are used to produce the best aggregate response for all of the processes being controlled.

In [Dean and Boddy, 1988], we define a framework for constructing solutions to time-dependent planning and control problems called *expectation-driven iterative refinement*. A solution to a time-dependent problem using expectation-driven iterative refinement will consist of a set of anytime algorithms and a *deliberation-scheduling* algorithm that allocates computational resources to the set of anytime algorithms based on expectations regarding their performance. An *optimal* deliberation schedule for a given situation is a deliberation schedule that maximises the expected utility of the robot's performance in that situation. An optimal deliberation-scheduling algorithm always generates the optimal schedule for the current situation. An optimal deliberation-scheduling algorithm thus provides the "principled way" of allocating resources that is needed. The basic idea is akin to using a domain-independent planning algorithm coupled with a domain-specific library of plans to generate sequences of actions in novel situations.

The expected utility of the anytime algorithms to be scheduled are represented by *performance profiles* that indicate how the expected utility of the answers returned by a given anytime algorithm changes with the amount of time allocated. Figure 2 shows performance profiles for two different algorithms, one for problems of type a, the other for problems of type b. Figure 3 shows two observations and the corresponding predicted events. In this case, all of the time between E1 and E2 can be used in computing a response for E2. If the expected utility of deliberat-

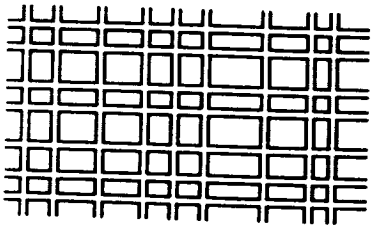

Figure 4: A city map for the robot-courier problem

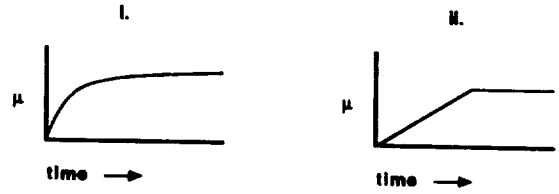

Figure 5: Performance profiles for the robot courier

ing further about E2 is higher than for spending time on E1, then time before E1 may be allocated to E2 as well. If E1 is of type a and E2 is of type b, then deliberation time will be allocated as shown by the shaded areas in Figure 3.

In the next section, we sketch an example of the application of expectation-driven iterative refinement to a robot-planning problem.

4 The Robot Courier

Suppose that we are in charge of designing the control program for a robot courier for a delivery service in a large city. The function of these couriers is to pick up small parcels and deliver them to specified locations. We assume that the city streets are arranged in an irregularly-spaced grid, and that the robot has a map of the city (see Figure 4) to assist in path planning. The robot is also capable of finding its way from one point to another without a planned path by keeping track of the heading of the destination as it performs a form of obstacle avoidance. Path planning helps because a planned path may be more direct. The utility of the robot's performance we define in terms of the time required to complete the entire set of deliveries.

The robot must plan a tour that visits all of the locations on its current list of deliveries. We refer to this as *tour improvement* planning. Once the robot has an ordering for the locations, it may spend time determining how to get from one to another of them. We refer to this as *path* planning. We assume that path planning is accomplished by constructing an ordered set of *target points* between the two locations. Arguably, controlling the robot in navigating between target points will not normally affect the expected utility of tour improvement or path planning. To simplify our discussion, we will concentrate on just these two types of planning and their role in controlling the behavior of the robot. Deliberation scheduling for the robot courier then consists of allocating time to algorithms for tour improvement and path planning based on the expected improvement in the robot's performance.

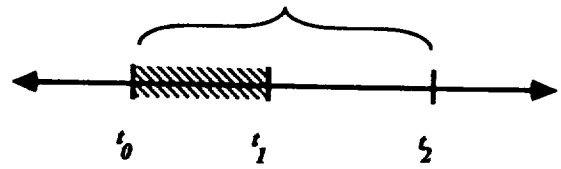

Figure 6: Path planning for a single path

In order to use expectation-driven iterative refinement, it is necessary that we have some expectations regarding the performance of our control algorithms. In the case of the robot-courier problem these expectations can be obtained by performing trial runs to gather the statistics necessary to construct performance profiles for the anytime algorithms for tour improvement and path planning. The tour-improvement algorithm we use is an adaptation of 2-OPT, and has a performance profile of the form shown in Figure 5-i. The path-planning algorithm we employ is a heuristic search algorithm of the sort described by Korf [Korf, 1987], and has a performance profile of the form shown in Figure 5-ii.

Consider the problem of scheduling just the path-planning algorithm for a tour whose order is already fixed. Since the utility of the robot's response is maximised by minimising the time expended in traversing the tour, the deliberation-scheduling algorithm should minimise the sum of planning and travel time required. Figure 6 shows a tour of two points (*i.e.*, one path to plan for). The robot plans from t_0 to t_1, and then spends from t_1 to t_2 traversing the path. The expected value of the distance from t_1 to t_2 will depend on how long the robot plans (*i.e.*, $t_1 - t_0$). The distance from t_0 to t_2 is the quantity to be minimized in order to produce an optimal deliberation schedule. The problem is slightly more complicated for a tour of n points. Figure 7 depicts the problem of deliberation scheduling for several points. There

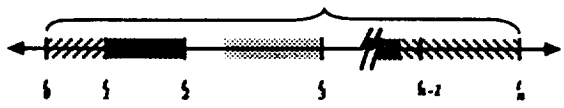

Figure 7: Path planning for several paths

are gaps where no planning is done, because all of the paths left to traverse have already been allocated the maximum useful deliberation time. The quantity to be minimized in this example is $\sum_{i=1}^{n}(t_i - t_{i-1})$. For the robot courier, this problem can be solved analytically; an optimal deliberation-scheduling algorithm appears in [Boddy and Dean, 1989].

Adding tour improvement complicates the problem. Since the path-planning algorithm requires a particular ordering of the points on the tour, the tour-improvement algorithm must be run first. Since the expected savings in time from path planning depends on the distance between locations, the expected utility of scheduling path planning depends on the expected length of the improved tour. In this case, the results of the two algorithms combine by composition: the expected utility of the final result involves the sum of the time spent on tour improvement and the time required to plan for and traverse the improved tour, which is a function of the time spent on tour improvement. It will probably help to go through this in a little more detail.

Figure 8 show a series of five snapshots illustrating the robot in various states of planning and deliberation scheduling. In each of the five snapshots, a "*" indicates the time at which the snapshot is taken, t_0 to t_1 is the time spent path planning before starting to travel to the first location in the current tour, and t_k to t_{k+1} (for $1 \leq k \leq n-1$) is the time spent traveling from the k-th to the $k+1$-st location. Figure 8-i depicts the situation in which the robot has some randomly-generated initial tour and λ_i is the expected time to traverse that tour. At this point the robot has to determine how to allocate time to tour improvement and path planning. The deliberation scheduling required to make this determination can be done very quickly using an algorithm discussed in [Boddy and Dean, 1989]. Here we assume that the time required for this type of deliberation scheduling is ϵ. The current framework for expectation-driven refinement requires that the time required for deliberation scheduling be negligible. In practice, the deliberation-scheduling algorithms we have implemented have been fast enough that this is a reasonable assumption.

Figure 8-ii shows the robot's expectations after the first bit of deliberation scheduling. The interval labeled δ is the amount of time allocated to tour improvement based on expectations concerning both the tour-improvement algorithm and the path-planning algorithm. Expectations regarding the path planner's performance are based on a tour in which the distance between any two adjacent locations is the same. The expected time spent in path planning and path traversal look something like λ_{ii}. Figure 8-iii shows the robot's expectations after actually performing tour improvement. At this point, the robot knows the exact order of the improved tour, and is no longer assuming that the distances are all the same. The interval labeled λ_{iii} is meant to indicate the expected time needed to traverse the tour with no path planning (t_0 is identical to t_1).

Now the robot must determine how to allocate time to planning each individual leg of the improved tour. This is deliberation scheduling of the sort depicted in Figure 7, in which the robot decides how long to apply the path planning algorithm to planning the route between each pair of adjacent locations in the tour. Figure 8-iv shows the resulting deliberation schedule after spending ϵ on this type of deliberation scheduling. The interval labeled λ_{iv} indicates the expected time for carrying out both path planning and path traversal. Finally, Figure 8-v shows the actual schedule and elapsed time λ_v resulting when the robot traverses the tour. Of course, the actual tour may take more or less time than the robot's initial expectations.

The robot-courier example illustrates both kinds of deliberation-scheduling interactions discussed earlier. Solving the problem as a whole requires solving two subproblems that compete for resources: tour improvement and path planning. Path planning for a tour requires dealing with multiple processes: planning the individual routes for each pair of adjacent locations in the tour.

5 Conclusion

The control of complex processes demands that we coordinate our computational and control processes to keep up with the processes that we seek to con-

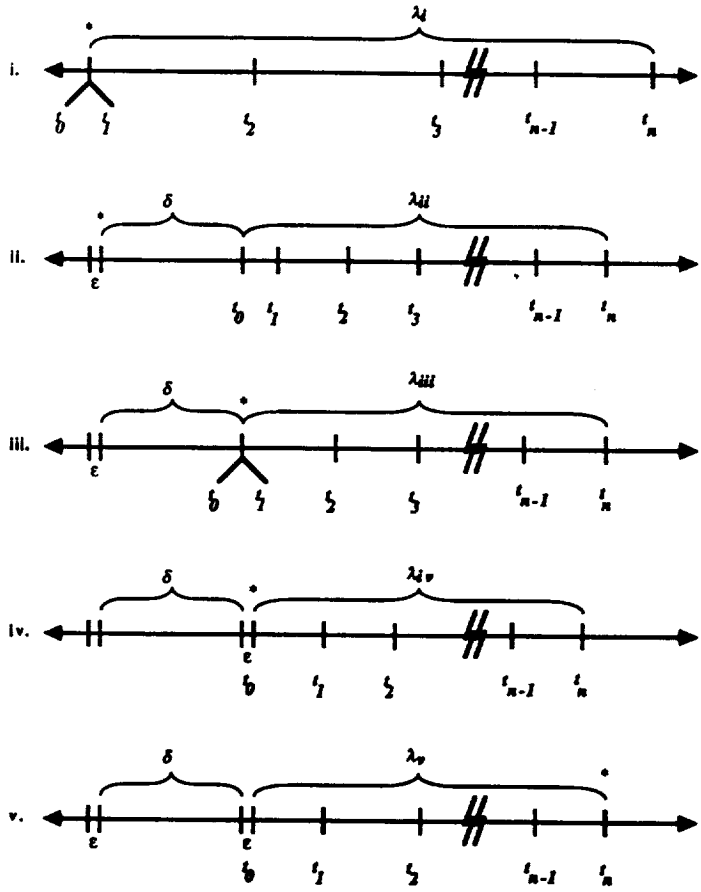

Figure 8: Combining tour improvement and path planning

trol. The traditional approach has been to try to make our computational processes so fast that we can keep pace with any process we are interested in controlling. However, as we tackle more and more complicated control problems, computational complexity limits our ability to reduce computing time. One way to deal with complexity is to use approximation schemes, sacrificing accuracy for speed. In situations in which the control processes provide varying amounts of time to respond, sticking to an approximation scheme with a fixed run time can result in a severe loss in performance. In this paper, we suggest a disciplined approach to using approximation algorithms to cope with processes whose critical or time-dependent events can be predicted with reasonable accuracy. Our approach enables us to allocate processor time to a set of approximation algorithms in order to optimize the performance of a complex control system. The framework of expectation-driven refinement described in this paper provides the basis for solving a wide variety of problems in control and process planning.

References

[Boddy and Dean, 1989] Mark Boddy and Thomas Dean. Solving time-dependent planning problems, 1989.

[Dean and Boddy, 1988] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *Proceedings AAAI-88*, pages 49–54. AAAI, 1988.

[Harel, 1987] David Harel. *ALGORITHMICS: The Spirit of Computing.* Addison-Wesley, 1987.

[Korf, 1987] Richard E. Korf. Real-time path planning. In *Proceedings DARPA Knowledge-Based Planning Workshop*, 1987.

[Lin and Kernighan, 1973] S. Lin and B. W. Kernighan. An effective heuristic for the travelling salesman problem. *Operations Research*, 21:498–516, 1973.