Anytime Problem Solving Using Dynamic Programming

Mark Boddy
Honeywell Systems and Research Center
MN 65-2100
3660 Technology Drive
Minneapolis, MN 55418
boddy@src.honeywell.com

Abstract

In previous work, we have advocated explicitly scheduling computation time for planning and problem solving (*deliberation*) using a framework called *expectation-driven iterative refinement*. Within this framework, we have explored the problem of allocating deliberation time when the procedures used for deliberation implement *anytime algorithms*: algorithms that return some answer for any allocation of time. In our search for useful techniques for constructing anytime algorithms, we have discovered that *dynamic programming* shows considerable promise for the construction of anytime algorithms for a wide variety of problems. In this paper, we show how dynamic programming techniques can be used to construct useful anytime procedures for two problems: multiplying sequences of matrices, and the Travelling Salesman Problem.

Dynamic programming can be applied to a wide variety of optimization and control problems, many of them relevant to current AI research (e.g., scheduling, probabilistic reasoning, and controlling machinery). Being able to solve these kinds of problems using anytime procedures increases the range of problems to which expectation-driven iterative refinement can be applied.

Introduction

In [Dean and Boddy, 1988], we advocate the practice of *deliberation scheduling*: the explicit allocation of computational resources for planning and problem-solving, based on expectations on future events and the effects of that computation (*deliberation*). In the same paper, we propose the use of *anytime algorithms*: algorithms that return an answer for any allocation of computation time. In subsequent work [Boddy and Dean, 1989, Boddy, 1991], we have explored the use of a particular framework for deliberation scheduling using anytime algorithms. In this framework, called *expectation-driven iterative refinement*, deliberation time is allocated using expectations on the effect on the system's behavior of time allocated to any of several anytime decision procedures. These expectations are cached in the form of *performance profiles*: graphs showing how some parameter of the answer returned is expected to change as more time is allocated to a procedure.

One of the questions that we have been asked repeatedly since we started this line of research concerns the use of anytime decision procedures: what evidence is there that useful anytime algorithms can be found for a sufficiently wide variety of computational tasks to make this approach interesting? A preliminary search of the computer science literature turned up a wide variety of algorithms or classes of algorithms that could be employed in anytime procedures. Among the kinds of algorithms we found:

- Numerical approximation – For example, Taylor series approximations (e.g., computing π or e) and iterative finite-element methods.
- Heuristic search – Algorithms for heuristic search, in particular those employing variable lookahead and fast evaluation functions, can be cast as anytime algorithms [Pearl, 1985, Korf, 1990].
- Probablistic algorithms – One family of probabilistic algorithms that can easily be adapted for anytime use are *Monte Carlo* algorithms [Harel, 1987].
- Probabilistic inference – A wide variety of methods has been developed for approximate evaluation of belief nets (i.e., providing bounds on the posterior distribution, rather than the exact distribution). Several of these methods are anytime algorithms in the sense that the bounds get smaller for additional iterations of the basic method, e.g., [Horvitz *et al.*, 1989, Henrion, 1986].
- Discrete or symbolic processing – Symbolic processing can be viewed as the manipulation of finite sets (of bindings, constraints, entities, etc.) [Robert, 1986]. Elements are successively added to or removed from a set representing an approximate answer so as to reduce the difference between that set and a set representing the correct answer.

Recent work suggests that existing anytime decision

procedures can be combined into procedures for solving more complex problems [Boddy, 1991].

In the process of looking for useful methods for constructing anytime algorithms, we have come to realize that *dynamic programming* [Bellman, 1957] might be employed as an anytime technique. This is potentially an important result: dynamic programming can be applied to a wide variety of optimization and control problems, many of them relevant to current AI research (e.g., scheduling, probabilistic reasoning, and controlling machinery). Being able to solve these kinds of problems using anytime procedures greatly increases the range of problems to which expectation-driven iterative refinement can easily be applied.

In this paper, we explore the use of dynamic programming algorithms in anytime decision procedures. In the next section we review dynamic programming. In subsequent sections we discuss the kinds of problems for which dynamic programming is best suited, with an emphasis on problems relevant to current research in AI, show how to construct anytime decision procedures using dynamic programming, and present some results regarding the behavior of the resulting procedures. The final section summarizes the main points of the paper and draws some conclusions.

Dynamic Programming

Dynamic programming is a methodology for the solution of problems that can be modelled as a sequence of decisions (alternatively, problems that can be broken into smaller problems whose results are then combined). For example, consider the problem of multiplying the following sequence of matrices:

$$M_1 = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{pmatrix}, \quad M_2 = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \end{pmatrix}$$

$$M_3 = \begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix}$$

Matrix multiplication is associative, so we can multiply M_1 and M_2, then multiply the result by M_3. Or we can multiply M_2 and M_3 first. The number of scalar multiplications required in each case is different. Multiplying M_1 by M_2 requires $4 * 2 * 4 = 32$ multiplications. Multiplying the resulting 4×4 matrix times M_3 requires an additional $4*4*4 = 64$ multiplications, for a total cost of 96. Multiplying M_2 and M_3 first results in a total cost of $(2 * 4 * 4) + (4 * 2 * 4) = 64$.

For longer sequences, the savings can be considerable. Using dynamic programming to solve this problem, we start by caching all the pairwise multiplication costs, then proceed to cache the cheapest way to multiply any three adjacent matrices together, then any four, and so on, each time using the results cached in preceding steps. The cost of finding the optimal answer in this way is $O(n^2)$ space and $O(n^3)$ time, where n is the number of matrices in the sequence.

More generally, dynamic programming is a methodology for solving *sequential decision problems*. Let $\mathcal{S}$ be a set of states, $\mathcal{D}$ the set of possible decisions, $R : \mathcal{S} \times \mathcal{D} \rightarrow \Re$ a *reward* function, and $\phi : \mathcal{S} \times \mathcal{D} \rightarrow \mathcal{S}$ a function mapping from the current state and a decision to the next state. The reward resulting in a single step from making decision d in state s_i is $R(s_i, d)$. The next state is $\phi(s_i, d)$. We call the sum of the rewards from a sequence of decisions the *value* of the sequence. The maximum possible value for one step starting in state s_i is

$$V_1(s_i) = \max_{d \in \mathcal{D}} R(s_i, d)$$

The value resulting from d depends on the decisions made in all the following states. Choosing the decision d that maximizes the value of the sequence of n states starting in s_i involves finding

$$V_n(s_i) = \max_{d \in \mathcal{D}} [R(s_i, d) + V_{n-1}(\phi(s_i, d))]$$

This can be solved, at least in principle. A very long or infinite sequence of decisions can be handled using *discounting*, in which the reward resulting from being in a given state is weighted inversely to how far in the future the state is. The resulting optimization problem is

$$V_n(s_i) = \max_{d \in \mathcal{D}} [R(s_i, d) + \alpha V_{n-1}(\phi(s_i, d))]$$

where $\alpha < 1$. We calculate an approximate answer, where the number of terms considered depends on α and the precision required.

Dynamic programming involves the computation of a *policy*: a specification of what decision to make in a given state so as to maximize the resulting value of a sequence of decisions. For dynamic programming to be useful, a sequential decision problem must obey Bellman's *principle of optimality:*

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision [Bellman, 1957].

For example, given a sequence of ten matrices to multiply together, the cost of multiplying the product of the first five and the product of the second five together does not depend on how those products were generated (i.e., how we associated the matrices in each group).

The problem is more complex when the outcome of a decision is uncertain. In this case, an optimal policy maximizes the *expected* value of a sequence of decisions. Let $P_{i,j}(d)$ be the probability of ending in state s_j, starting from s_i and making decision d. The recursive definition of the optimization problem for this case is

$$V_n(s_i) = \max_{d \in \mathcal{D}} [R(s_i, d) + \sum_{s_j \in \mathcal{S}} P_{i,j}(d) V_{n-1}(s_j)]$$

As long as the principle of optimality holds, standard dynamic programming techniques can be applied to solve stochastic problems.[1]

Dynamic Programming Applications

Dynamic programming can be applied to a wide range of problems. Any problem that can be cast as a sequential decision problem and that obeys the principle of optimality (or some extension thereof) is a candidate for a dynamic programming solution. Classes of problems for which dynamic programming solutions can frequently be found include scheduling and resource problems (e.g., inventory control, sequencing and synchronizing independent processes, network flow problems, airline scheduling, investment problems), control problems (e.g., optimal control, stochastic processes), and problems in game theory [Larson and Casti, 1978, Bellman, 1957].

In addition, several extant approaches to planning and problem-solving in AI have a dynamic-programming flavor to them. For example, the progressive construction of STRIPS triangle tables can be viewed as the successive construction of a policy: each entry caches an improved decision for a particular state [Fikes *et al.*, 1972]. Drummond and Bresina's [Drummond and Bresina, 1990] anytime *synthetic projection* is related to triangle tables, and has an even stronger dynamic programming flavor. In their work, simple causal rules corresponding to system actions and other events are manipulated to construct *situated control rules* (SCRs) that can be used by an agent interacting with the world. These SCRs are iteratively constructed as the result of additional search and the prior construction of other SCRs. The *cut-and-commit* strategy they employ to direct the search for new SCRs is similar to problem decomposition techniques for dynamic programming—though the principle of optimality does not appear to hold. A paradigm for planning suggested by Stan Rosenschein [Rosenschein, 1989] called *domains of competence* involves a technique, related to synthetic projection, in which the system iteratively expands the set of states from which it knows how to achieve a given goal state.

Anytime Dynamic Programming

Anytime algorithms tend to be iterative algorithms. Using dynamic programming, we achieve this iteration through the successive caching of more and more complete answers. Each subproblem solved can reduce the space that must be searched to find an optimal answer. If an anytime decision procedure using dynamic programming is required to supply an answer before it has completed, some inexpensive (and suboptimal) method is employed to choose an answer from the remaining possibilities. Two reasonable alternatives are to choose randomly (to make the remaining decisions at random) or to use some form of greedy algorithm.[2]

In this section, we present the results of implementing anytime decision procedures using dynamic programming for two examples: the matrix-multiplication problem described previously, and the TSP.

```
Procedure: Build_dp_table(seq)
  begin
      n := length(seq)
      if n >= 3 then
         for size = 1 to n
             for i = 1 to n - size + 1
                 j := i + size - 1
                 Find_min_cost(⟨s_i, ..., s_j⟩)
  end

Procedure: Find_min_cost(seq)
begin
  k := length(seq)
  if k = 1 then
     return {0, φ}
  else if k = 2 then
     return {N_1 * M_1 * M_2, φ}
  else
     {cost, assoc} := Lookup_dp_entry(seq)
     if cost >= 0 then
        return {cost, assoc}
     else
        c_min := ∞, a_min := φ
        for i = 1 to k - 1
           {c, a} := Find_min_cost(⟨s_1, ..., s_i⟩)+
                     Find_min_cost(⟨s_{i+1}, ..., s_k⟩)
           c := c + (N_1 * M_i * M_k)
           if c < c_min then
              c_min := c
              a_min := {{i, a}}
        Make_dp_entry(seq, c_min, a_min)
        return {c_min, a_min}
end
```

Figure 1: Dynamic programming for matrix association

Matrix Multiplication Revisited

In the section on dynamic programming we showed how, given a sequence of matrices to multiply, the number of scalar multiplications necessary depended on the order in which the matrices were multiplied together. We also sketched a dynamic-programming solution to finding a minimum-cost way of combining a given sequence of matricies. The procedure Build_dp_table in

[1] Recent work extends the application of dynamic programming to stochastic decision problems that do not satisfy the principle of optimality [Carraway *et al.*, 1989].

[2] A greedy algorithm makes decisions so as to obtain the best answer possible in one step. A greedy algorithm for the Travelling Salesman Problem might successively add to a partial tour, choosing at each step the location minimizing the length of the resulting partial tour.

```
Procedure: Random_search(seq)
 begin
   k := length(seq)
   if k = 1 then
      return {0, φ}
   else if k = 2 then
      return {N1 * M1 * M2, φ}
 else
    {cost, assoc} := Lookup_dp_entry(seq)
    if cost >= 0 then
        return {cost, assoc}
    else
       k := length(seq)
       i := random(1, k - 1)
       {c, a} := Random_search(⟨s1, ..., si⟩) +
                 Random_search(⟨si+1, ..., sk⟩)
       c := c + (N1 * Mi * Mk)
       amin := {{i, a}}
       return {c, amin}
 end
```

Figure 2: Random search for matrix association

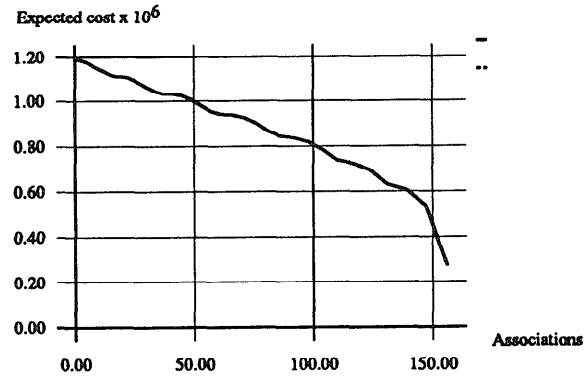

Figure 3: Expected cost as a function of work done

Figure 1 implements that solution. The procedure Find_min_cost adds the table entries and returns two values: the cost of the optimal way of associating the (sub)sequence of matrices, and the optimal association itself. The notation N_i (alt. M_i) denotes the number of rows (columns) in the ith element of the sequence of matrices $seq = \langle s_1, \ldots, s_n \rangle$. The function Lookup_dp_entry looks in the table for the sequence it is passed. If the sequence is found, the optimal cost and association are returned. If the sequence is not found, a cost of -1 is returned. The associations are recursively constructed by keeping track of the value of i (the point at which to divide the current subsequence) resulting in the minimum cost for constructing and combining subsequences. Build_dp_table iterates over subsequences so that when the procedure is looking for optimal associations for subsequences of length k, the optimal associations for all subsequences of length less than k have already been computed. This keeps the recursion in Find_min_cost to a maximum depth of 2, and ensures that only at the top level is any search required—every subsequence of length greater than 3 is already in the table.

Each additional result cached (each call to Make_dp_entry) provides more information regarding an optimal solution. Intuitively, it seems reasonable that more information should make it easier to generate a good solution by inexpensive means. This intuition is borne out experimentally. We repeatedly generated sequences of 10 matrices with dimensions randomly chosen from the interval $[1, 100]$. For each sequence, a dynamic programming solution was generated one step at a time, each step consisting of calculating and storing the optimal way to multiply some subsequence of size k. After each step, the average cost of the solution that would be generated by a random search procedure was calculated.

The search procedure is given in Figure 2. This procedure works recursively by breaking the current sequence into two pieces at a random point, finding the cost of multiplying the resulting subsequences, and adding the cost of combining their products. If a cached answer is found for a particular subsequence that answer is used, otherwise the procedure bottoms out at pairwise multiplications. The cost of running this procedure is $O(n)$, where n is the number of matrices.

Figure 3 is the result of 500 trials of the experiment described above. The x axis is the number of associations that have been considered. We use this rather than the number of cached answers because the work needed to compute the optimal association for a subsequence depends on its length: a subsequence of length 3 requires checking 2 alternatives, while finding the optimal answer for the full sequence of length 10 requires checking 9 possible ways of combining subsequences. The y axis is the average number of scalar multiplications required for an association chosen by Random_search, given the answers cached so far. The periodic plateaus are steps at which k changes. Apparently, having the first cached answer for a subsequence of size k does not help as much as adding answers for additional k-length subsequences. The big drop at the end is the result of going from randomly choosing among nine possibilities, one of which is optimal, to knowing the optimal answer.

An anytime procedure for doing matrix multiplication using these procedures calls Build_dp_table. If interrupted before completion, the procedure returns the results of calling Random_search on the entire sequence. Random_search uses the information cached in the calls to Make_dp_entry. The longer Build_dp_table runs, the more information is available, and the better the expected cost of the answer will be. Calling

```
Procedure: TSP_dp(locs)
  begin
    n := length(locs)
    for size = 3 to n − 1
        while pts := Get_next_point_set(size)
            for pt in pts
                Add_one_subtour(pt, pts)
    Add_one_subtour(l1, locs)
  end

Procedure: Add_one_subtour(end, pts)
begin
  lmin := ∞
  for e in pts − {end}
      s := Get_relevant_subtour(e, pts − {end})
      l := subtour_length(s) + dist(e, end)
      if l < lmin then
          lmin := l
          smin := s
   make_dp_entry(lmin, concat(subtour_pts(smin), ⟨end⟩))
end
```

Figure 4: Dynamic programming for the TSP

Random_search after an answer is requested imposes a delay on producing that answer equal to the cost of running Random_search. In the worst case, when Build_dp_table has not run at all, the cost is $O(n)$. If this delay is unacceptable, Random_search can be run periodically through the $O(n^3)$ iterations necessary to construct the complete dynamic programming solution.

Travelling Salesman

An instance of the TSP specifies a set of locations and a set of costs (distance, time, fuel, etc.) to move between them. The problem is to minimize the cost incurred in visiting every location exactly once, returning to the location we start at. In the example discussed in this section, the locations are points in a convex subset of the real plane, and inter-location cost is simply the distance between them. Even with these restrictions, the problem is still NP-complete.

The procedure TSP_dp in Figure 4 constructs a dynamic-programming solution for a TSP instance by caching the optimal tour ordering for successively larger subsets of the set of locations given in the problem instance. For each subset, the procedure caches the optimal ordering for a tour starting at l_1, for any endpoint within the subset not equal to l_1 (except for the final call to Add_one_subtour, which finds the optimal ordering for the complete tour). Repeated calls to the function Get_next_point_set result in enumerating all the subsets of a given size of the set of locations $locs - l_1$. Add_one_subtour loops through all the possible subtours (tours whose points are exactly those in $pts - \{end\}$), and finds the minimum-length ordering for the current tour. The procedure

```
Procedure: Greedy_tour(locs)
 begin
   t := ⟨l1⟩
   pts := locs − l1
   while pts
       s := Find_approp_subtour(t)
       if s then
           return concat(t, subtour_pts(s))
       else
           imin := arg min_j dist(lj, last(t))
           t := concat(t, ⟨lj⟩)
           pts := pts − lj
   return t
 end
```

Figure 5: Greedy construction of a tour

Get_relevant_subtour(end, pts) finds the cached optimal subtour for the set of locations pts, starting at l_1 and ending at end. This can be made a constant-time operation, as can the procedure Get_next_point_set.

To implement an anytime procedure, we also need the procedure Greedy_Tour in Figure 5. At each step, Greedy_Tour chooses the location closest to the endpoint of the tour it has constructed so far. If at any point it can find a subtour in the table that has the same endpoint and includes all the tour points it has not yet used, the procedure uses that subtour to complete the tour ordering. Finding a cached subtour (the procedure Find_approp_subtour) can be done in constant time. The time cost of running Greedy_tour is $O(n^2)$, where n is the number of points in the tour.

The anytime procedure runs TSP_dp until it is interrupted, then runs Greedy_tour. The longer TSP_dp has run, the more likely Greedy_tour is to find a useful subtour. Just as in the anytime procedure for matrix multiplication, this procedure runs the relatively inexpensive ($O(n^2)$ vs. 2^n) suboptimal procedure once it has been interrupted. As before, the delay could be removed, at the cost of running the greedy procedure periodically during the construction of the optimal tour.

We performed a series of 50 experiments. In each one, a random set of 11 points was generated. We then ran TSP_dp on the set of points, keeping track of the cost of the tour generated using Greedy_tour after each iteration. Figure 6 graphs the average cost of the solution found by Greedy_tour over the total number of iterations performed in the loop in Add_one_subtour. It is clear from the graph that the average cost of the solution drops in a reasonably well-behaved way with increasing iterations of TSP_dp. It is worth noting that the expected cost of the tour obtained using the greedy algorithm alone is only about 12% worse than the optimal tour.

Summary and Conclusions

In this paper, we have shown how dynamic programming techniques can be used to construct useful any-

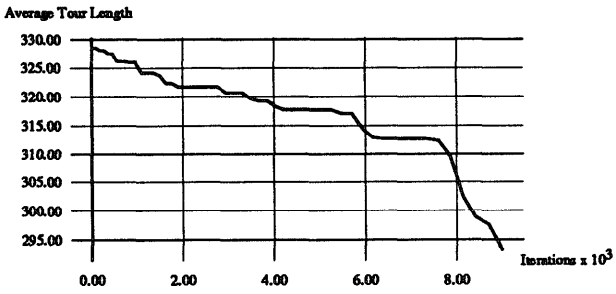

Figure 6: Expected savings over answers cached

time procedures for two problems: multiplying sequences of matrices, and the Travelling Salesman Problem. In each case, the procedure iteratively constructs a dynamic programming solution. If the procedure is interrupted, it uses an inexpensive alternate procedure to construct a solution, making use of the partial solution it has constructed so far.

Finding a good alternate procedure is important. Using a procedure that does not make effective use of the partial dynamic programming solution results in poor anytime performance: the expected value of the answers returned tends to be very low until the dynamic programming solution is completed or nearly completed. For example, our first attempt at writing Greedy_tour searched forward from the end of the longest optimal tour found so far, rather than back from the end until a cached tour is encountered. Despite the fact that this seems intuitively to be a better use of the cached answers, the resulting anytime procedure performed abysmally.

There is another sense in which dynamic programming can be viewed as an iterative technique that we have not discussed in this paper. *Policy iteration* involves the successive approximation of an optimal policy. This requires that we repeatedly calculate (an approximation to) an entire policy, and is thus unlikely to provide a useful basis for anytime algorithms. Barto and Sutton [Barto *et al.*, 1989] discuss the use of policy iteration in the incremental construction of a policy for controlling a dynamical system as more information becomes available over time.

Dynamic programming can be applied to a wide variety of problems. We have shown that this includes a range of problems of interest in AI, including scheduling, resource, and control problems. The work presented in this paper suggests how to go about generating anytime procedures for solving some of these problems.

Acknowledgements

Jack Breese first pointed out to me the importance of dynamic programming and the principle of optimality in the analysis of sequential decision problems. Tom Dean made the connection between the construction of SCRs in [Drummond and Bresina, 1990] and dynamic programming. Bob Schrag and two anonymous reviewers provided useful comments.

References

Andrew G. Barto, R.S. Sutton, and C.J.C.H Watkins. Learning and sequential decision making. Technical Report 89-95, University of Massachusetts at Amherst Department of Computer and Information Science, 1989.

R.E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *IJCAI89*, 1989.

Mark Boddy. Solving time-dependent problems: A decision-theoretic approach to planning in dynamic environments. Technical Report CS-91-06, Brown University Department of Computer Science, 1991.

R. L. Carraway, T. L. Morin, and H. Moskowitz. Generalized dynamic programming for stochastic combinatorial optimization. *Operations Research*, 37(5):819–829, 1989.

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

Mark Drummond and John Bresina. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 138–144, 1990.

Richard Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

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

M. Henrion. Propagating uncertainty by logic sampling in bayes' networks. In *Proceedings of the Second Workshop on Uncertainty in Artificial Intelligence*, 1986.

E. J. Horvitz, H. J. Suermondt, and G. F. Cooper. Bounded conditioning: Flexible inference for decisions under scarce resources. In *Proceedings of the Fifth Workshop on Uncertainty in Artificial Intelligence*, 1989.

Richard Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2):189–212, 1990.

Robert E. Larson and John L. Casti. *Principles of Dynamic Programming, Part I.* Marcell Dekker, Inc., New York, New York, 1978.

Judea Pearl. *Heuristics.* Addison-Wesley, 1985.

F. Robert. *Discrete Iterations: A Metric Study.* Springer-Verlag, 1986.

Stan Rosenschein. *Personal communication.* 1989.