Journal of Heuristics, 3: 245–262 (1997)

Rollout Algorithms for Combinatorial Optimization*

DIMITRI P. BERTSEKAS, JOHN N. TSITSIKLIS AND CYNARA WU
Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA 02139

Abstract

We consider the approximate solution of discrete optimization problems using procedures that are capable of magnifying the effectiveness of any given heuristic algorithm through sequential application. In particular, we embed the problem within a dynamic programming framework, and we introduce several types of rollout algorithms, which are related to notions of policy iteration. We provide conditions guaranteeing that the rollout algorithm improves the performance of the original heuristic algorithm. The method is illustrated in the context of a machine maintenance and repair problem.

Key Words:

1. Introduction

We discuss the approximate solution of broad classes of combinatorial optimization problems by embedding them within a Dynamic Programming framework (DP for short). The key idea is to employ a given heuristic in the construction of an optimal cost-to-go function approximation, which is then used in the spirit of the Neuro-Dynamic Programming/Reinforcement Learning methodology (NDP for short; see (Barto, Bradtke and Singh (1995), Bertsekas and Tsitsiklis (1996)) for broad discussions of this methodology).

In the next section, we will introduce a general graph search problem that will serve as the context of our methodology. To illustrate the ideas involved, however, let us consider the following type of problem, which includes as special cases problems such as shortest path, assignment, scheduling, matching, etc. The problem is characterized by a finite set U of feasible solutions, and by a cost function $g(u)$. Each solution u has N components; that is, it has the form $u = (u_1, u_2, \ldots, u_N)$, where N is a positive integer. We want to find a solution $u \in U$ that minimizes $g(u)$.

We can view the preceding problem as a sequential decision problem, whereby the components $u_1, \ldots, u_N$ are selected one-at-a-time. An n-tuple $(u_1, \ldots, u_n)$ consisting of the first n components of a solution is called an *n-solution*. We associate n-solutions with the nth stage of a DP problem. In particular, for $n = 1, \ldots, N$, the states of the nth stage are of the form $(u_1, \ldots, u_n)$. The initial state is a dummy (artificial) state. From this state we may move to any state (u_1), with u_1 belonging to the set

$$U_1 = \{\bar{u}_1 \mid \text{there exists a solution of the form } (\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_N) \in U\}.$$

*Research supported by NSF under Grant DMI-9625489.

More generally, from a state of the form

$$(u_1, \ldots, u_{n-1}),$$

we may move to any state of the form

$$(u_1, \ldots, u_{n-1}, u_n),$$

with u_n belonging to the set

$$U_n(u_1, \ldots, u_{n-1}) = \{\bar{u}_n \mid \text{there exists a solution of the form } (u_1, \ldots, u_{n-1}, \bar{u}_n, \ldots, \bar{u}_N) \in U\}.$$

The controls available at state $(u_1, \ldots, u_{n-1})$ are $u_n \in U_n(u_1, \ldots, u_{n-1})$. The terminal states of the problem correspond to the N-solutions $(u_1, \ldots, u_N)$, and the only nonzero cost is the terminal cost $g(u_1, \ldots, u_N)$.

Let $J^*(u_1, \ldots, u_n)$ denote the optimal cost starting from the n-solution $(u_1, \ldots, u_n)$, that is, the optimal cost of the problem over solutions whose first n components are constrained to be equal to $u_i, i = 1, \ldots, n$, respectively. If we knew the optimal cost-to-go function $J^*(u_1, \ldots, u_n)$, we could construct an optimal solution by a sequence of N single component minimizations. In particular, an optimal solution $(u_1^*, \ldots, u_N^*)$ could be obtained through the algorithm

$$u_i^* = \arg \min_{u_i \in U_i(u_1^*, \ldots, u_{i-1}^*)} J^*(u_1^*, \ldots, u_{i-1}^*, u_i), \quad i = 1, \ldots, N. \tag{1}$$

Unfortunately, the preceding DP formulation is seldom viable, because of the prohibitive computation required to obtain the optimal cost-to-go function $J^*(u_1, \ldots, u_n)$. In NDP, this difficulty is dealt with by replacing $J^*(u_1, \ldots, u_n)$ with approximations

$$\tilde{J}(u_1, \ldots, u_n),$$

and by obtaining a suboptimal solution $(\tilde{u}_1, \ldots, \tilde{u}_N)$ sequentially, through the algorithm

$$\tilde{u}_i = \arg \min_{u_i \in U_i(\tilde{u}_1, \ldots, \tilde{u}_{i-1})} \tilde{J}(\tilde{u}_1, \ldots, \tilde{u}_{i-1}, u_i), \quad i = 1, \ldots, N. \tag{2}$$

The function $\tilde{J}$ will be called a *scoring function* or *approximate cost-to-go function*, and may contain some adjustable parameter vector that can be tuned using special "training" methods. In this paper, however, we restrict attention to scoring functions that are based on heuristic algorithms. In particular, we will assume that we have a heuristic algorithm, which starting from an n-solution $(u_1, \ldots, u_n)$, can produce a complete N-solution $(u_1, \ldots, u_N)$ whose cost is denoted by $H(u_1, \ldots, u_n)$. One possibility, studied in this paper, is to approximate the optimal cost-to-go function with the scoring function

$$\tilde{J}(u_1, \ldots, u_n) = H(u_1, \ldots, u_n). \tag{3}$$

A more general possibility is to use multiple heuristic algorithms, which are weighted with some scalar weights to provide the approximation $\tilde{J}(u_1, \ldots, u_n)$. In this paper, we assume that the weights are fixed (although they could be adjusted through a separate trial-and-error process). In a more general NDP approach, the weights could be tunable parameters and could depend on some features of the given problem. This more general approach will be the subject of a separate report.

In the next section, we consider a graph search problem that is more general than the combinatorial problem described above, and we introduce a corresponding DP framework. We then formulate several sequential methods for constructing solutions, and we illustrate these methods through some examples.

2. Graph search problems and rollout algorithms

Let us introduce a graph search problem that can serve as a general model for discrete optimization. We are given a directed graph with node set $\mathcal{N}$ and arc set $\mathcal{A}$, and a special node s, which we call the *origin*. We are also given a subset $\bar{\mathcal{N}}$ of nodes, called *destinations*, and a cost function $g(i)$ on the set $\bar{\mathcal{N}}$. The destination nodes are terminal in the sense that they have no outgoing arcs. We allow the node and arc sets, $\mathcal{N}$ and $\mathcal{A}$, to contain an infinite number of elements. We require, however, that the number of destination nodes be finite. We want to find a directed path that starts at the origin s, ends at one of the destination nodes $i \in \bar{\mathcal{N}}$, and is such that the cost $g(i)$ is minimized.

For convenience, and without loss of generality, we will assume that given an ordered pair of nodes (i, j), there is at most one arc with start node i and end node j, which (if it exists) will be denoted by (i, j). In this way, a directed path consisting of arcs $(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)$ is unambiguously specified as the sequence of nodes $(i_1, i_2, \ldots, i_n)$.

As an example of the preceding formulation, consider the optimization problem discussed in the preceding section. The origin is an artificial starting state, the n-solutions $(u_1, \ldots, u_n)$, $n = 1, \ldots, N$, can be identified with the remaining nodes, and the (complete) N-solutions can be identified with the set of destinations.

Similar to the construction used in the preceding section, we can transform the graph search problem into a DP problem. In particular, the nodes correspond to the states of the DP problem, the controls available at a given state/node and the corresponding successor states/nodes are the outgoing arcs from the node and the associated end nodes of the arcs, respectively. The destination nodes i are terminal states of the DP problem, where the terminal cost $g(i)$ is incurred.

Let us now assume that we have a path construction algorithm $\mathcal{H}$, which given a non-destination node $i \notin \bar{\mathcal{N}}$, constructs a directed path $(i, i_1, \ldots, i_m, \bar{i})$ starting at i and ending at one of the destination nodes $\bar{i}$. Implicit in this assumption is that for every non-destination node, there exists at least one path starting at that node and ending at some destination node. We denote by $H(i)$ the corresponding cost; that is,

$$H(i) = g(\bar{i}), \quad \forall\, i \notin \bar{\mathcal{N}}. \tag{4}$$

If i is a destination node, by convention we write

$$H(i) = g(i), \quad \forall i \in \bar{\mathcal{N}}. \tag{5}$$

Note that while the algorithm $\mathcal{H}$ will generally yield a suboptimal solution, the path that it constructs may involve a fairly sophisticated suboptimization. For example, $\mathcal{H}$ may construct several paths ending at destination nodes according to some heuristics, and then select the path that yields minimal cost.

One possibility for suboptimal solution of the problem is to start at the origin s and use the algorithm $\mathcal{H}$ to obtain a solution of cost $H(s)$. We instead propose to use $\mathcal{H}$ to construct a path to a destination node sequentially. At the typical step of the sequence, we consider all downstream neighbors j of a node i, we run $\mathcal{H}$ starting from each of these neighbors, and we then move to the neighbor from which $\mathcal{H}$ gives the best result. The idea of starting with some algorithm, and using it to construct another, hopefully improved, algorithm is implicit in the policy iteration method of DP and in the use of a rollout policy, which is a form of policy iteration; see (Bertsekas and Tsitsiklis (1996)) (the name "rollout policy" was used by Tesauro (Tesauro and Galperin (1996)) in connection with one of his simulation-based computer backgammon algorithms). This connection will be shown to be particularly relevant to our context, and for this reason we call the sequential version of $\mathcal{H}$ the *rollout algorithm based on* $\mathcal{H}$, and we denote it by $\mathcal{RH}$. We note that the idea of sequential selection of candidates for participation in a solution is implicit in several combinatorial optimization contexts. For example this idea is embodied in the sequential fan candidate list strategy as applied in tabu search (see Glover, Taillard and de Werra (1993)). This idea is also used in a manner similar to the present paper in the sequential automatic test procedures of Pattipati (see e.g., Pattipati and Alexandridis (1990)).

To formally describe the rollout algorithm, let $N(i)$ denote the set of downstream neighbors of node i, that is,

$$N(i) = \{j \mid (i, j) \text{ is an arc}\}. \tag{6}$$

Note that $N(i)$ is nonempty for every non-destination node i, since there exists at least one path starting at i and ending at a destination. The rollout algorithm starts with the origin node s. At the typical step, given a node sequence $(s, i_1, \ldots, i_m)$, where i_m is not a destination, $\mathcal{RH}$ adds to the sequence a node i_{m+1} such that

$$i_{m+1} = \arg \min_{j \in N(i_m)} H(j). \tag{7}$$

If i_{m+1} is a destination node, the path $(s, i_1, \ldots, i_m, i_{m+1})$ is taken to be the solution generated by $\mathcal{RH}$, with corresponding cost $g(i_{m+1})$. Otherwise, the process is repeated with the sequence $(s, i_1, \ldots, i_m, i_{m+1})$ replacing $(s, i_1, \ldots, i_m)$. Once $\mathcal{RH}$ has terminated with a path $(s, i_1, \ldots, i_m, \bar{i})$, we will have obtained the paths constructed by $\mathcal{H}$ starting from each of the nodes $i_1, \ldots, i_m$. The best of these paths yields a cost

$$\min_{k=1,\ldots,m} H(i_k).$$

We first note that while $\mathcal{H}$, by definition, has the property that it yields a path terminating at a destination starting from any node, the rollout algorithm $\mathcal{RH}$ need not have this property in the absence of additional conditions. We will later introduce a variant of $\mathcal{RH}$ that always terminates. The following example illustrates how $\mathcal{RH}$ may fail to terminate.

Example 1 (Nonterminating $\mathcal{RH}$). Assume that there is a single destination d and that all other nodes are arranged in a directed cycle. Each non-destination node i has two outgoing arcs: one arc that belongs to the cycle, and another arc which is (i, d). Suppose that starting from a node $i \neq d$, the path generated by $\mathcal{H}$ consists of two arcs: the first arc is (i, j) where j is the node subsequent to i on the cycle, and the second arc is (j, d). Then it can be seen that $\mathcal{RH}$ continually repeats the cycle and never terminates.

We say that $\mathcal{RH}$ is *terminating* if it is guaranteed to terminate finitely starting from any node. One important case where $\mathcal{RH}$ is terminating is *when the graph is acyclic and the set of nodes $\mathcal{N}$ is finite*, since then the nodes of the path generated by $\mathcal{RH}$ cannot be repeated and their number is bounded by the number of nodes in $\mathcal{N}$. As a first step towards developing another case where $\mathcal{RH}$ is terminating, we introduce the following definition.

Definition 1. The algorithm $\mathcal{H}$ is said to be *sequentially consistent* if for every node i, whenever $\mathcal{H}$ generates the path $(i, i_1, \ldots, i_m, \bar{i})$ starting at i, it also generates the path $(i_1, \ldots, i_m, \bar{i})$ starting at the node i_1.

Example 1 above illustrates a situation where $\mathcal{H}$ is not sequentially consistent. On the other hand, there are many examples of sequentially consistent algorithms that are used as heuristics in combinatorial optimization. For instance, *greedy algorithms* of various types and other algorithms that inherently have a sequential character often tend to be sequentially consistent. The following example provides an important context where a sequentially consistent algorithm arises.

Example 2 ($\mathcal{H}$ defined by a heuristic evaluation function). Suppose that we have a real-valued function F defined on $\mathcal{N}$, where $F(i)$ represents an estimate of the optimal cost starting from i, that is, the minimal cost $g(\bar{i})$ that can be obtained with a path that starts at i and ends at one of the destination nodes $\bar{i} \in \bar{\mathcal{N}}$. Then F can be used to define the path generating algorithm $\mathcal{H}$ as follows:

The algorithm $\mathcal{H}$ starts at a node i with the degenerate path (i). At the typical step, given a path $(i, i_1, \ldots, i_m)$, where i_m is not a destination, $\mathcal{RH}$ adds to the path a node i_{m+1} such that

$$i_{m+1} = \arg \min_{j \in N(i_m)} F(j). \tag{8}$$

If i_{m+1} is a destination, $\mathcal{H}$ terminates with the path $(s, i_1, \ldots, i_m, i_{m+1})$. Otherwise, the process is repeated with the path $(s, i_1, \ldots, i_m, i_{m+1})$ replacing $(s, i_1, \ldots, i_m)$.

Let us assume that $\mathcal{H}$ terminates starting from every node (this has to be verified independently). Let us also assume that whenever there is a tie in the minimization of Eq. (8), the algorithm $\mathcal{H}$ resolves the tie in a manner that is fixed and independent of the starting

node i of the path, e.g., by resolving the tie in favor of the numerically smallest node j that attains the minimum in Eq. (8). Then it can be seen that $\mathcal{H}$ is sequentially consistent.

For a sequentially consistent algorithm $\mathcal{H}$, we will assume a restriction in the way the algorithm $\mathcal{RH}$ resolves ties in selecting the next node on its path via Eq. (7); this restriction will guarantee that $\mathcal{RH}$ is terminating, and is also needed to ensure that $\mathcal{RH}$ is sequentially consistent. We will assume that whenever there is a tie in the minimization (7), $\mathcal{RH}$ resolves the tie in a manner that is independent of the starting node of the path (similar to the preceding example). To elaborate, suppose that at the typical step, where we are given a node sequence $(s, i_1, \ldots, i_m)$, we have

$$H(i_m) = \min_{j \in N(i_m)} H(j). \tag{9}$$

In this case, the path $(i_m, i'_{m+1}, \ldots, \bar{i}')$ generated by the algorithm $\mathcal{H}$ starting at i_m yields a cost $H(i_m) = g(\bar{i}')$ that is equal to the best obtainable from the successor nodes $i \in N(i_m)$, and the node i'_{m+1} attains the minimum in the preceding equation. We require that if there are some other nodes, in addition to i'_{m+1}, attaining this minimum, the next node added to the current sequence $(s, i_1, \ldots, i_m)$ is i'_{m+1}. Under this convention for tie-breaking, we show in the following proposition that $\mathcal{RH}$ terminates at a destination and yields a cost that is no larger than the cost yielded by $\mathcal{H}$.

Proposition 1. *Let the algorithm $\mathcal{H}$ be sequentially consistent. Then $\mathcal{RH}$ is terminating. Furthermore, if $(i_1, \ldots, i_m)$ is the path generated by $\mathcal{RH}$ starting from a non-destination node i_1 and ending at a destination node i_m we have*

$$H(i_1) \geq H(i_2) \geq \cdots \geq H(i_{m-1}) \geq H(i_m). \tag{10}$$

Equivalently, in view of Eq. (7), we have

$$H(i_m) = \min \left\{ H(i_1), \min_{j \in N(i_1)} H(j), \ldots, \min_{j \in N(i_{m-1})} H(j) \right\}. \tag{11}$$

Proof: Let $(i_1, i_2, \ldots, i_m, \ldots)$ be the path generated by $\mathcal{RH}$ starting from a non-destination node i_1. For each $m = 1, 2, \ldots$, let $(i_m, i'_{m+1}, i'_{m+2}, \ldots, \bar{i}_m)$ be the path generated by $\mathcal{H}$ starting at i_m, where $\bar{i}_m$ is a destination node. Then, since $\mathcal{H}$ is sequentially consistent, we have

$$H(i_m) = H(i'_{m+1}) = g(\bar{i}_m). \tag{12}$$

Furthermore, since $i'_{m+1} \in N(i_m)$, we have using the definition of $\mathcal{RH}$ [cf. Eq. (7)]

$$H(i'_{m+1}) \geq \min_{j \in N(i_m)} H(j) = H(i_{m+1}).$$

Combining the last two relations, we obtain

$$H(i_m) \geq H(i_{m+1}), \quad m = 1, 2, \ldots \tag{13}$$

and also, equivalently,

$$g(\bar{i}_m) \geq g(\bar{i}_{m+1}), \quad m = 1, 2, \ldots \tag{14}$$

To show that $\mathcal{RH}$ is terminating, consider two successive nodes i_m and i_{m+1} generated by $\mathcal{RH}$. Then, in view of Eq. (13), either $H(i_m) > H(i_{m+1})$, or else $H(i_m) = H(i_{m+1})$. In the latter case, in view of the convention for breaking ties that occur in Eq. (9), the path generated by $\mathcal{H}$ starting from i_{m+1} is the tail portion of the path generated by $\mathcal{H}$ starting from i_m, and has one arc less. Thus the number of nodes generated by $\mathcal{RH}$ between successive times that the inequality $H(i_m) > H(i_{m+1})$ holds is finite. On the other hand, the inequality $H(i_m) > H(i_{m+1})$ can occur only a finite number of times, since the number of destination nodes is finite, and the destination node of the path generated by $\mathcal{H}$ starting from i_m cannot be repeated if the inequality $H(i_m) > H(i_{m+1})$ holds. Therefore, $\mathcal{RH}$ is terminating. The relation (13) then implies the desired relations (10) and (11), thus completing the proof. □

Proposition 1 shows that in the sequentially consistent case, algorithm $\mathcal{RH}$ has an important "automatic cost sorting" property, whereby it follows the best path generated by $\mathcal{H}$. In particular, when $\mathcal{RH}$ generates a path $(i_1, \ldots, i_m)$, it does so by using $\mathcal{H}$ to generate a collection of other paths starting from all the successor nodes of the intermediate nodes $i_1, \ldots, i_{m-1}$. However, $(i_1, \ldots, i_m)$ is guaranteed to be the best among this collection [cf. Eq. (11)]. Of course this does not guarantee that the path generated by $\mathcal{RH}$ will be a near-optimal path, because the collection of paths generated by $\mathcal{H}$ may be "poor". Still, the property whereby $\mathcal{RH}$ at all times follows the best path found so far is intuitively reassuring.

The following example illustrates the preceding concepts.

Example 3 (One-dimensional walk). Consider a person who walks on a straight line and at each time period takes either a unit step to the left or a unit step to the right. There is a cost function assigning cost $g(i)$ to each integer i. Given an integer starting point on the line, the person wants to minimize the cost of the point where he will end up after a given and fixed number N of steps.

We can formulate this problem as a graph search problem of the type discussed in the preceding section. In particular, without loss of generality, let us assume that the starting point is the origin, so that the person's position after n steps will be some integer in the interval $[-n, n]$. The nodes of the graph are identified with pairs (k, m), where k is the number of steps taken so far ($k = 1, \ldots, N$) and m is the person's position ($m \in [-k, k]$). A node (k, m) with $k < N$ has two outgoing arcs with end nodes $(k+1, m-1)$ (corresponding to a left step) and $(k+1, m+1)$ (corresponding to a right step). The starting state is $(0, 0)$ and the terminating states are of the form (N, m), where m is of the form $N - 2l$ and $l \in [0, N]$ is the number of left steps taken.

Let $\mathcal{H}$ be defined as the algorithm, which, starting at a node (k, m), takes $N - k$ successive steps to the right and terminates at the node $(N, m + N - k)$. Note that $\mathcal{H}$ is sequentially

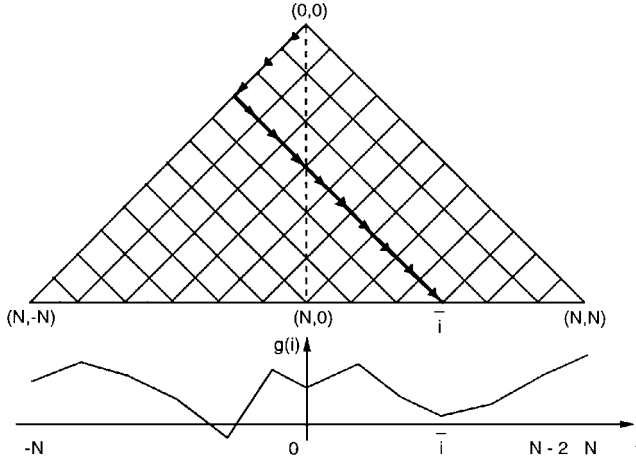

Figure 1. Illustration of the path generated by the rollout algorithm $\mathcal{RH}$ in Example 3. $\mathcal{RH}$ keeps moving to the left up to the time where $\mathcal{H}$ generates two destinations $(N, \bar{i})$ and $(N, \bar{i}-2)$ with $g(\bar{i}) \leq g(\bar{i}-2)$. Then it continues to move to the right ending at the destination $(N, \bar{i})$, which corresponds to the local minimum closest to N.

consistent. The algorithm $\mathcal{RH}$, at node (k, m) compares the cost of the destination node $(N, m+N-k)$ (corresponding to taking a step to the right and then following $\mathcal{H}$) and the cost of the destination node $(N, m+N-k-2)$ (corresponding to taking a step to the left and then following $\mathcal{H}$). Let us say that an integer $i \in [-N+2, N-2]$ is a *local minimum* if $g(i-2) \geq g(i)$ and $g(i) \leq g(i+2)$. Let us also say that N (or $-N$) is a local minimum if $g(N-2) > g(N)$ [or $g(-N) \leq g(-N+2)$, respectively]. Then it can be seen, using Eq. (11), that starting from the origin (0, 0), $\mathcal{RH}$ obtains the local minimum that is closest to N, (see figure 1). This is no worse (and typically better) than the integer N obtained by $\mathcal{H}$. Note that if g has a unique local minimum in the set of integers in the range $[-N, N]$, the minimum must also be global, and it will be found by $\mathcal{RH}$. This example illustrates how $\mathcal{RH}$ may exhibit "intelligence" that is totally lacking from $\mathcal{H}$, and is in agreement with the result of Proposition 1.

3. Interpretation in terms of DP and policy iteration

Let us now interpret the concepts and results presented so far in the context of DP. If we view the graph search problem of this section as a DP problem in the manner described earlier, we can see that the algorithm $\mathcal{H}$ corresponds to a policy μ_H, that is, a choice at any one node of a successor node, which may depend on the choice of initial node/state. In particular, if a path $(i_1, \ldots, i_m, i_{m+1})$ is generated by $\mathcal{H}$ starting from node i_1, then for any $i_k, k = 1, \ldots, m$, the policy μ_H specifies the successor node choice

$$\mu_H(i_1, i_k) = i_{k+1}.$$

In the terminology of DP, such a policy is called *semi-Markov*. On the other hand, if $\mathcal{H}$ is sequentially consistent, the choice of the successor node does not depend on the initial node/state, and in the terminology of DP, μ_H is called a *Markov* or *stationary* policy.

Consider now the rollout algorithm $\mathcal{RH}$, assuming that it is terminating. Then it can be seen that similar to $\mathcal{H}$, algorithm $\mathcal{RH}$ defines a policy μ_{RH} that is *stationary* regardless of whether μ_H is stationary. Thus, in particular, $\mathcal{RH}$ is sequentially consistent (compare also with Example 2). In fact it can be verified that μ_{RH} is the policy that would be generated by a single iteration of the classical policy iteration algorithm starting with policy μ_H. It is well-known from DP theory that a policy iteration starting from a terminating stationary policy produces another terminating stationary policy of improved cost. This is in agreement with the result of Proposition 1, which essentially shows that if μ_H is stationary, then μ_{RH} is stationary and has improved cost.

Let us note that, assuming $\mathcal{RH}$ is terminating, we may consider the rollout algorithm $\mathcal{RH}$, in place of $\mathcal{H}$. This will generate another algorithm, call it $\mathcal{R}^2\mathcal{H}$, which in a DP context will correspond to a policy μ_{R^2H}. This is the stationary policy obtained from μ_{RH} via a policy iteration, or equivalently, from μ_H via two successive policy iterations.

Finally, let us consider a two-step lookahead rollout algorithm, which we will call $\mathcal{R}_2\mathcal{H}$. This algorithm is defined similar to $\mathcal{RH}$ with the only difference that at a given node i, we consider the set $N_2(i)$ of all possible *two-step successor nodes of* i, that is, the set of nodes j for which there exists an intermediate node j' such that $j' \in N(i)$ and $j \in N(j')$. The next node generated by $\mathcal{R}_2\mathcal{H}$ is a node $\bar{j}$ such that

$$\bar{j} = \arg \min_{j \in N_2(i)} H(j). \tag{15}$$

The algorithm $\mathcal{R}_2\mathcal{H}$ bears no clear relation to algorithms $\mathcal{RH}$ and $\mathcal{R}^2\mathcal{H}$. In particular, no inference can be drawn regarding the cost functions of these three algorithms, other than the relation mentioned earlier that $\mathcal{R}^2\mathcal{H}$ yields no worse cost than $\mathcal{RH}$ starting from any initial node.

4. Alternative rollout algorithms

We now consider some generalizations of the results and algorithms discussed so far. We first show that the result of Proposition 1 holds under weaker conditions on the algorithm $\mathcal{H}$. Let us introduce the following definition:

Definition 2. Suppose that algorithm $\mathcal{H}$ generates, starting at each node $i \notin \bar{\mathcal{N}}$, a path $(i, i_1, \ldots, i_m, \bar{i})$ with the property

$$H(i) \geq H(i_1). \tag{16}$$

Then the algorithm $\mathcal{H}$ is said to be *sequentially improving*.

It can be seen that a sequentially consistent $\mathcal{H}$ is also sequentially improving, with equality holding in Eq. (16). If we now use Eq. (16) in place of Eq. (12) in the proof of

Proposition 1, we see that this proof carries through verbatim. We thus have the following generalization of Proposition 1:

Proposition 2. *Let the algorithm $\mathcal{H}$ be sequentially improving, and suppose that $\mathcal{RH}$ is terminating. Then, if $(i_1, \ldots, i_m)$ is the path generated by $\mathcal{RH}$ starting from a non-destination node i_1 and ending at a destination node i_m, we have*

$$H(i_m) = \min \left\{ H(i_1), \min_{j \in N(i_1)} H(j), \ldots, \min_{j \in N(i_{m-1})} H(j) \right\}. \tag{17}$$

Example 4. Consider the one-dimensional walk problem of Example 3, and let $\mathcal{H}$ be defined as the algorithm that, starting at a node (k, m), compares the cost $g(m + N - k)$ (corresponding to taking all of the remaining $N - k$ steps to the right) and the cost $g(m - N + k)$ (corresponding to taking all of the remaining $N - k$ steps to the left), and accordingly moves to node

$$(N, m + N - k) \quad \text{if } g(m + N - k) \leq g(m - N + k),$$

or to node

$$(N, m - N + k) \quad \text{if } g(m - N + k) < g(m + N - k).$$

It can be seen that $\mathcal{H}$ is not sequentially consistent, but is instead sequentially improving. Using Eq. (17), it follows that starting from the origin (0, 0), $\mathcal{RH}$ obtains the global minimum of g in the interval $[-N, N]$, while $\mathcal{H}$ obtains the better of the two points $-N$ and N.

4.1. The extended rollout algorithm

We can always modify the problem and the algorithm $\mathcal{H}$ so that Proposition 2 applies. In particular, let us consider the *extended version of the problem*, whereby the graph $(\mathcal{N}, \mathcal{A})$ is enlarged by adding for each non-destination node i an arc $(i, d(i))$, where $d(i)$ is the destination at which the path generated by $\mathcal{H}$ terminates, starting from i. (This arc is not added if it already exists.) Then $\mathcal{H}$ is modified so that starting from each non-destination node i for which

$$\min_{j \in N(i)} H(j) > H(i) = g(d(i)), \tag{18}$$

it generates instead the path $(i, d(i))$. It is seen that the modified version of $\mathcal{H}$ so obtained, referred to as the *extended* $\mathcal{H}$ and denoted by $\mathcal{H}_e$, is sequentially improving. Thus, Proposition 2 applies to the rollout algorithm based on the extended $\mathcal{H}$, which is referred to as the *extended rollout algorithm* and is denoted by $\mathcal{RH}_e$. This algorithm proceeds exactly like $\mathcal{RH}$ up to the first node i for which Eq. (18) holds, and then terminates with the destination node $d(i)$.

4.2. *The optimized rollout algorithm*

If $\mathcal{H}$ is not sequentially improving, it is possible in general that $\mathcal{RH}$ generates a worse solution than the solutions generated by $\mathcal{H}$ from the same starting node. However, it is always possible to correct this deficiency by a minor modification of $\mathcal{RH}$. In particular, in the process of running $\mathcal{RH}$, one generates several solutions, and upon termination of $\mathcal{RH}$, one can choose out of all these solutions, one that has minimal cost. This version of the rollout algorithm, is referred to as the *optimized rollout algorithm* and is denoted by $\mathcal{R}^*\mathcal{H}$. Note that if $\mathcal{H}$ is terminating, then $\mathcal{R}^*\mathcal{H}$ is guaranteed to generate a no worse solution than all of the algorithms $\mathcal{H}$, $\mathcal{RH}$, and $\mathcal{RH}_e$.

4.3. *The fortified rollout algorithm*

Let us introduce an alternative sequential version of $\mathcal{H}$. This version is referred to as the *fortified rollout algorithm* and is denoted by $\mathcal{R}\bar{\mathcal{H}}$. As the notation suggests, $\mathcal{R}\bar{\mathcal{H}}$ turns out to be the rollout algorithm based on a path construction algorithm $\bar{\mathcal{H}}$, which is derived from $\mathcal{H}$ and will be defined shortly. The fortified rollout algorithm $\mathcal{R}\bar{\mathcal{H}}$ starts at s, and maintains, in addition to the current sequence of nodes $(s, i_1, \ldots, i_m)$, a path

$$P(i_m) = (i_m, i'_{m+1}, \ldots, i'_k), \tag{19}$$

ending at a destination i'_k. Initially, $P(s)$ is the path generated by $\mathcal{H}$ starting from s. At the typical step of $\mathcal{R}\bar{\mathcal{H}}$, we have a node sequence $(s, i_1, \ldots, i_m)$, where $i_m \notin \bar{\mathcal{N}}$, and the path $P(i_m) = (i_m, i'_{m+1}, \ldots, i'_k)$. Then, if

$$\min_{j \in N(i_m)} H(j) < g(i'_k), \tag{20}$$

$\mathcal{R}\bar{\mathcal{H}}$ adds to the node sequence $(s, i_1, \ldots, i_m)$ the node

$$i_{m+1} = \arg \min_{j \in N(i_m)} H(j),$$

and sets $P(i_{m+1})$ to the path generated by $\mathcal{H}$, starting from i_{m+1}. On the other hand, if

$$\min_{j \in N(i_m)} H(j) \geq g(i'_k), \tag{21}$$

$\mathcal{R}\bar{\mathcal{H}}$ adds to the node sequence $(s, i_1, \ldots, i_m)$ the node

$$i_{m+1} = i'_{m+1},$$

and sets $P(i_{m+1})$ to the path $(i_{m+1}, i'_{m+2}, \ldots, i'_k)$. If i_{m+1} is a destination, $\mathcal{R}\bar{\mathcal{H}}$ terminates, and otherwise $\mathcal{R}\bar{\mathcal{H}}$ starts the next step with $m+1$ replacing m.

The main idea behind the construction of $\mathcal{R}\bar{\mathcal{H}}$ is to follow the path $P(i_m)$ unless a path of lower cost is discovered through Eq. (20). It can be seen that $\mathcal{R}\bar{\mathcal{H}}$ may be viewed as the

rollout algorithm $\mathcal{RH}$ corresponding to a modified version of $\mathcal{H}$, called *fortified* $\mathcal{H}$, and denoted $\bar{\mathcal{H}}$. This algorithm is applied to a slightly modified version of the original problem, which involves an additional downstream neighbor for each node i_m that is generated in the course of the algorithm $\mathcal{R}\bar{\mathcal{H}}$ and for which the condition (21) holds. For every such node i_m, the additional neighbor is a copy of i'_{m+1}, and the path generated by $\bar{\mathcal{H}}$ starting from this copy is $(i'_{m+1}, \ldots, i'_k)$. From every other node, the path generated by $\bar{\mathcal{H}}$ is the same as the path generated by $\mathcal{H}$. It can be seen that $\bar{\mathcal{H}}$ is sequentially improving, so that $\mathcal{R}\bar{\mathcal{H}}$ is terminating and has the automatic cost sorting property of Propositions 1 and 2; that is,

$$H(i_m) = \min\left\{H(i_1), \min_{j\in N(i_1)} H(j), \ldots, \min_{j\in N(i_{m-1})} H(j)\right\}.$$

The above property can also be easily verified directly, using the definition of $\mathcal{R}\bar{\mathcal{H}}$. It can also be seen that the fortified rollout algorithm $\mathcal{R}\bar{\mathcal{H}}$ will always perform at least as well as the extended rollout algorithm $\mathcal{RH}_e$. The potential improvement in performance is obtained at the expense of the modest additional overhead involved in maintaining the path $P(i_m)$. Note that when $\mathcal{H}$ is sequentially consistent, all three rollout algorithms $\mathcal{RH}$, $\mathcal{RH}_e$, and $\mathcal{R}\bar{\mathcal{H}}$ coincide.

4.4. Using multiple path construction algorithms

We note that one may use multiple path construction algorithms in the preceding framework. In particular, let us assume that we have K algorithms $\mathcal{H}_1, \ldots, \mathcal{H}_K$. The kth of these algorithms, given a node $i \notin \mathcal{N}$, produces a path $(i, i_1, \ldots, i_m, \bar{i})$ that ends at a destination node $\bar{i}$, and the corresponding cost is denoted by $H_k(i) = g(\bar{i})$. Generalizing our earlier approach, we can use the K algorithms in an approximation architecture of the form

$$\tilde{J}(i) = \min_{k=1,\ldots,K} H_k(i), \tag{22}$$

or of the form

$$\tilde{J}(i, r_1, \ldots, r_K) = \sum_{k=1}^{K} r_k H_k(i), \tag{23}$$

where r_k are some fixed scalar weights obtained by trial and error. The rollout algorithms $\mathcal{RH}$, $\mathcal{RH}_e$, and $\mathcal{R}\bar{\mathcal{H}}$ easily generalize for the case of Eq. (22), by replacing $H(i)$ with $\tilde{J}(i)$, and by defining the path generated starting from a node i as the path generated by the path construction algorithm which attains the minimum in Eq. (22). In the case of Eq. (23), the rollout algorithm $\mathcal{RH}$ also generalizes easily by replacing $H(i)$ with $\tilde{J}(i, r_1, \ldots, r_K)$, but in order to generalize the algorithms $\mathcal{RH}_e$, and $\mathcal{R}\bar{\mathcal{H}}$, the path generated from a node i must also be defined. There are several possibilities along this line. A different type of possibility for the case of Eq. (23), is to use tunable weights, which are obtained by training using NDP methodology. This is discussed in the recent textbook (Bertsekas and Tsitsiklis (1996)), and will be the subject of a future report.

4.5. *Extension for intermediate transition costs*

Finally, let us consider a problem where in addition to the terminal cost $g(i)$, there is a cost $c(i, j)$ for a path to traverse an arc (i, j). Within this context, the cost of a path $(i_1, i_2, \ldots, i_n)$ that starts at i_1 and ends at a destination node i_n is redefined to be

$$g(i_n) + \sum_{k=1}^{n-1} c(i_k, i_{k+1}). \tag{24}$$

One way to transform this problem into one involving a terminal cost only is to redefine the graph of the problem so that nodes correspond to sequences of nodes in the original problem graph. Thus if we have arrived at node i_k using path $(i_1, \ldots, i_k)$, the choice of i_{k+1} as the next node is viewed as a transition from state $(i_1, \ldots, i_k)$ to state $(i_1, \ldots, i_k, i_{k+1})$. Both states $(i_1, \ldots, i_k)$ and $(i_1, \ldots, i_k, i_{k+1})$ are viewed as nodes of a redefined graph. Furthermore, in this redefined graph, a destination node has the form $(i_1, i_2, \ldots, i_n)$, where i_n is a destination node of the original graph, and has a cost given by Eq. (24).

After the details are worked out, we see that to recover our earlier algorithms and analysis, we need to modify the cost of the heuristic algorithm $\mathcal{H}$ as follows: If the path $(i_1, \ldots, i_n)$ is generated by $\mathcal{H}$ starting at i_1, then

$$H(i_1) = g(i_n) + \sum_{k=1}^{n-1} c(i_k, i_{k+1}).$$

Furthermore, the rollout algorithm $\mathcal{RH}$ at node i_m selects as next node i_{m+1} the node

$$i_{m+1} = \arg \min_{j \in N(i_m)} [c(i_m, j) + H(j)];$$

[cf. Eq. (7)]. The definition of a sequentially consistent algorithm remains unchanged. Furthermore, Proposition 1 remains unchanged except that Eqs. (10) and (11) are modified to read

$$H(i_k) \geq c(i_k, i_{k+1}) + H(i_{k+1}) = \min_{j \in N(i_k)} [c(i_k, j) + H(j)], \quad k = 1, \ldots, m-1.$$

A sequentially improving algorithm should now be characterized by the property

$$H(i_k) \geq c(i_k, i_{k+1}) + H(i_{k+1})$$

if i_{k+1} is the next node on the path generated by $\mathcal{H}$ starting from i_k. Furthermore, Proposition 2 remains unchanged except that Eq. (17) is modified to read

$$H(i_k) \geq \min_{j \in N(i_k)} [c(i_k, j) + H(j)], \quad k = 1, \ldots, m-1.$$

Finally, the criterion $\min_{j\in N(i_m)} H(j) < g(i'_k)$ [cf. Eq. (20)] used in the fortified rollout algorithm, given the sequence $(s, i_1, \ldots, i_m)$, where $i_m \notin \bar{\mathcal{N}}$, and the path $P(i_m) = (i_m, i'_{m+1}, \ldots, i'_k)$, should be replaced by

$$\min_{j\in N(i_m)} \big[c(i_m, j) + H(j)\big] < g(i'_k) + c(i_m, i'_{m+1}) + \sum_{l=m+1}^{k-1} c(i'_l, i'_{l+1}).$$

5. Some computational experience

We have tested rollout algorithms in a variety of contexts. We have consistently found that they can be very effective and that they can substantially improve the performance of the original heuristic. In this section, we provide an example involving a combinatorial two-stage maintenance and repair problem (in fact a stochastic programming problem).

Consider a repair shop that has a number of spare parts that can be used to maintain a given collection of machines of T different types over two stages. A machine that is broken down at the beginning of a stage can be immediately repaired with the use of one spare part or it can be discarded, in which case a cost C_t is incurred. A machine of type t that is operational (possibly thanks to repair) at the beginning of a stage breaks down during that stage with probability p_t, independently of other breakdowns, and may be repaired at the end of the stage, so that it is operational at the beginning of the next stage. Knowing the number of available spare parts, number of machines of each type, and the number of initially broken down machines, the problem is to find the repair policy that minimizes the expected total cost of the machines that break down and do not get repaired. The essence of the problem is to trade off repairing the first stage breakdowns with leaving enough spare parts to repair the most expensive of the second stage breakdowns.

Let s be the number of initially available spare parts, and let m and y be the vectors

$$m = (m_1, \ldots, m_T), \quad y = (y_1, \ldots, y_T),$$

where $m_t, t = 1, \ldots, T$, is the number of machines of type t (all assumed to be initially working), and $y_t, t = 1, \ldots, T$, is the number of breakdowns of machines of type t during the first stage. The decision to be made is

$$u = (u_1, \ldots, u_T),$$

where u_t is the number of spare parts used to repair breakdowns of machines of type t at the end of the first stage. We note that at the second stage, it is optimal to use the remaining spare parts to repair the machines that break in the order of their cost (that is, repair the most expensive broken machines, then if spare parts are left over, consider the next most expensive broken machines, etc). Thus, if we start the second stage with $\bar{s}$ spare parts, and $\bar{m}_t$ working machines of type $t = 1, \ldots, T$, and during the second stage, $\bar{y}_t$ machines of type t break, $t = 1, \ldots, T$, the optimal cost of the second stage, which is denoted by

$G(\bar{m}, \bar{y}, \bar{s})$, where

$$\bar{m} = (\bar{m}_1, \ldots, \bar{m}_T), \quad \bar{y} = (\bar{y}_1, \ldots, \bar{y}_T),$$

can be calculated analytically. We will not give the formula for the function G, because it is quite complicated, although it can be easily programmed for computation.

Let us denote by R the expected value, over the second stage breakdowns, of the second stage cost

$$R(\bar{m}, \bar{s}) = E_{\bar{y}}[G(\bar{m}, \bar{y}, \bar{s})].$$

Then in the first stage, and once the first stage breakdowns are known, the problem is to find $u = (u_1, \ldots, u_T)$ that solves the problem

$$\text{minimize } \sum_{t=1}^{T} C_t(y_t - u_t) + R\left(m - y + u, \sum_{t=1}^{T} u_t\right)$$

$$\text{subject to } \sum_{t=1}^{T} u_t \le s, \quad 0 \le u_t \le y_t, \quad t = 1, \ldots, T.$$

This is the problem we wish to solve approximately by using a rollout algorithm.

We reformulate this first stage problem as a path construction problem. In the reformulated problem, the nodes of the graph are triplets (m, y, s). Destination nodes are the ones for which $y = 0$ and the repair/no repair decision has been made for all the first stage breakdowns. At a non-destination node, the control choices are to select a particular breakdown type, say t, with $y_t > 0$, and then select between two options:

(1) Leave the breakdown unrepaired, in which case the triplet (m, y, s) evolves to

$$(m_1, \ldots, m_{t-1}, m_t - 1, m_{t+1}, \ldots, m_T, y_1, \ldots, y_{t-1}, y_t - 1, y_{t+1}, \ldots, y_T, s)$$

and the cost C_t of permanently losing the corresponding machine is incurred.

(2) Repair the breakdown, in which case the triplet (m, y, s) evolves to

$$(m_1, \ldots, m_T, y_1, \ldots, y_{t-1}, y_t - 1, y_{t+1}, \ldots, y_T, s - 1),$$

and no cost is incurred.

Once we have $y_1 = \cdots = y_T = 0$, there is no decision to make, and we simply pay the optimal cost-to-go of the second stage, $R(\bar{m}_1, \ldots, \bar{m}_T, \bar{s})$, and terminate.

We consider rollout policies based on heuristic algorithms. We used the following two heuristics, which given the triplet (m, y, s), produce a first stage solution u:

(1) *Proportional heuristic*: In this heuristic, we compute an estimate $\bar{N}$ of the total number of second stage breakdowns based on the probabilities p_t of breakdown of individual

machines of type t. In particular, we have $\bar{N} = \sum_{t=1}^{T} p_t N_t$, where N_t is a heuristic estimate of the number of working machines of type t at the start of the second stage, based on the already known vectors m and y. We form the estimated ratio of first stage to second stage breakdowns,

$$f = \frac{\sum_{t=1}^{T} y_t}{\bar{N}}.$$

We then fix the number of spare parts to be used in the first stage to

$$s_1 = afs,$$

where a is a positive parameter. The first stage problem is then solved by allocating the s_1 spare parts to machines of type t in the order of the costs $C_t(1 - p_t)$. (The factor of $1 - p_t$ is used to account for the undesirability of repairing machines that are likely to break again.) The constant a provides a parametrization of this heuristic. In particular, when $a < 1$, the heuristic is conservative, allocating more spare parts to the second stage than the projected ratio of breakdowns suggests, while if $a > 1$, the heuristic is more myopic, giving higher priority to the breakdowns that have already occurred in the first stage.

(2) *Value-based heuristic*: In this heuristic, given the state, we assign values of C_t and $C_t(1 - p_t)$ to each spare part used to repair a machine of type t in the second stage and the first stage, respectively. Note that a repair of a machine in the first stage is valued less than a repair of the same machine in the second stage, since a machine that is repaired in the first stage may break down in the second stage and require the use of an extra spare part. We rank-order the values C_t and $C_t(1 - p_t)$, $t = 1, \ldots, T$, and we repair broken down machines in decreasing order of value, using the estimate $p_t(m_t - y_t)$ for the number of machines to break down in the second stage.

We have tested the four rollout algorithms $\mathcal{RH}$, $\mathcal{R}^*\mathcal{H}$, $\mathcal{RH}_e$, and $\mathcal{R}\bar{\mathcal{H}}$ based on single and multiple heuristics, and we have compared their performance with the one of the corresponding heuristics, as well as with the optimal performance. Table 1 summarizes our results on a set consisting of 5000 randomly generated test problems. In these problems, there were 5 machine types, with costs 2, 4, 6, 8, and 10, respectively. The number of machines of each type was randomly chosen from the range [0, 10], the number of spare parts was randomly chosen from 0 to the total number of machines, and the breakdown probability for each machine type was randomly chosen from the range [0, 1]. A uniform distribution was used in each random choice. The optimal cost, averaged over the test sample of 5000 problems was calculated (by brute force) to be 33.69.

The proportional heuristic was used with three different values of the parameter a (0.5, 1.0, and 1.5), and the corresponding values are indicated in the 1st column of Table 1. Also, when multiple heuristics were used, they were combined into a single heuristic using the minimum cost formula (22). Thus for example, the heuristic Value/Prop ($a = 0.5$), consists of starting at a given node, running the value heuristic and the proportional heuristic with $a = 0.5$, and then out of the two paths generated, choosing the one with minimal cost.

Table 1. Test results on a set consisting of 5000 randomly generated test problems. Each row corresponds to a single heuristic algorithm or a combination of heuristic algorithms (this is the algorithm $\mathcal{H}$) as indicated in the leftmost entry. The second entry of the row gives the average cost over the test set corresponding to $\mathcal{H}$ starting from the initial node of the problem. Entries 3–6 in each row give the average cost over the test set for the corresponding rollout algorithms. The optimal cost, averaged over the test sample, is 33.69. The last entry gives the percentage gain in the error from optimality achieved by the best of the rollout algorithms, relative to the original heuristic [for example the heuristic Prop($a = 0.5$) of the 1st row is suboptimal by $49.00 - 33.69 = 15.31$, and the best rollout algorithm reduces this to $43.06 - 33.69 = 9.37$, resulting in a gain of $(15.31 - 9.37)/15.31$ or 38.8%].

Heuristic	$\mathcal{H}$	$\mathcal{RH}$	$\mathcal{R}^*\mathcal{H}$	$\mathcal{RH}_e$	$\mathcal{R}\bar{\mathcal{H}}$	% Gain
Prop($a = 0.5$)	49.00	43.09	43.06	44.13	44.10	38.80
Prop($a = 1.0$)	37.23	37.76	35.83	36.14	36.04	39.54
Prop($a = 1.5$)	41.28	36.11	34.93	35.04	34.98	83.66
Value	38.75	35.97	35.94	35.98	35.90	56.32
Value/Prop($a = 0.5$)	38.74	35.95	35.93	35.95	35.90	56.23
Value/Prop($a = 1.0$)	36.55	35.93	35.29	35.42	35.23	46.15
Value/Prop($a = 1.5$)	37.39	35.62	34.93	35.01	34.90	67.30
Prop($a = 0.5$)/Prop($a = 1.0$)	37.03	37.76	35.73	36.14	36.05	38.92
Prop($a = 0.5$)/Prop($a = 1.5$)	38.98	36.18	35.04	35.14	35.12	74.29
Prop($a = 1.0$)/Prop($a = 1.5$)	36.61	36.13	34.93	35.13	34.97	57.53
Prop($a = 0.5$)/Prop($a = 1.0$)/ Prop($a = 1.5$)	36.40	36.14	34.94	35.14	34.99	53.87
Value/Prop($a = 1.0$)/Prop($a = 1.5$)	36.20	35.57	34.85	35.01	34.81	55.38
Value/Prop($a = 0.5$)/Prop($a = 1.0$)	36.55	35.93	35.29	35.42	35.24	45.80
Value/Prop($a = 0.5$)/Prop($a = 1.5$)	37.38	35.62	34.93	35.00	34.90	67.21

It can be seen that the rollout algorithms can improve significantly the performance of the original heuristic algorithm $\mathcal{H}$. In particular, the relative improvement (the percentage reduction of the deviation from optimality, given in the last entry in each row of Table 1) is significant. Furthermore, in agreement with the earlier analysis, it can be seen that:

(a) All of the algorithms $\mathcal{R}^*\mathcal{H}$, $\mathcal{RH}_e$, and $\mathcal{R}\bar{\mathcal{H}}$ consistently outperform the original heuristic algorithm $\mathcal{H}$. On the other hand, because $\mathcal{H}$ is not guaranteed to be sequentially improving, the standard rollout algorithm $\mathcal{RH}$ may perform worse than the original heuristic $\mathcal{H}$ (see the 2nd and 8th rows of the table).

(b) The optimized rollout algorithm $\mathcal{R}^*\mathcal{H}$ consistently outperforms the standard and the extended rollout algorithms $\mathcal{RH}$ and $\mathcal{RH}_e$.

(c) The fortified rollout algorithm $\mathcal{R}\bar{\mathcal{H}}$ consistently outperforms the standard and the extended rollout algorithms $\mathcal{RH}$ and $\mathcal{RH}_e$. On the other hand there is no clear superiority relation between the optimized and the fortified algorithms.

References

Barto, A.G., S.J. Bradtke, and S.P. Singh. (1995). "Learning to Act Using Real-Time Dynamic Programming." *Artificial Intelligence* 72, 81–138.

Barto, A.S. and R. Sutton. (1997). *Reinforcement Learning*. MIT Press (forthcoming).

Bertsekas, D.P. and J.N. Tsitsiklis. (1996). *Neuro-Dynamic Programming*. Belmont, MA: Athena Scientific.

Glover, F., E. Taillard, and D. de Werra. (1993). "A User's Guide to Tabu Search." *Annals of Operations Research* 41, 3–28.

Pattipati, K.R. and M.G. Alexandridis. (1990). "Application of Heuristic Search and Information Theory to Sequential Fault Diagnosis." *IEEE Transactions on Systems, Man, and Cybernetics* 20, 872–887.

Tesauro, G. and G.R. Galperin. (1996). "On-Line Policy Improvement Using Monte Carlo Search." Unpublished report.