

Discounted Deterministic Markov Decision Processes and Discounted All-Pairs Shortest Paths

Omid Madani *

Mikkel Thorup †

Uri Zwick ‡

Abstract

We present two new algorithms for finding optimal strategies for discounted, infinite-horizon, Deterministic Markov Decision Processes (DMDP). The first one is an adaptation of an algorithm of Young, Tarjan and Orlin for finding minimum mean weight cycles. It runs in $O(mn + n^2 \log n)$ time, where n is the number of vertices (or states) and m is the number of edges (or actions). The second one is an adaptation of a classical algorithm of Karp for finding minimum mean weight cycles. It runs in $O(mn)$ time. The first algorithm has a slightly slower worst-case complexity, but is faster than the first algorithm in many situations. Both algorithms improve on a recent $O(mn^2)$ -time algorithm of Andersson and Vorobyov. We also present a randomized $\tilde{O}(m^{1/2}n^2)$ -time algorithm for finding Discounted All-Pairs Shortest Paths (DAPSP), improving several previous algorithms.

1 Introduction

Markov decision processes (MDPs) provide a mathematical model for sequential decision making under uncertainty. They are widely used to model stochastic optimization problems in various areas ranging from operations research, machine learning, artificial intelligence, economics and game theory. The study of MDPs started with the seminal work on Bellman [2]. (In some disciplines, MDPs are referred to as *discrete stochastic dynamic programming problems*.) More general *stochastic games* were previously considered by Shapley [28]. Influential books on MDPs were written by Howard [15] and Derman [9]. For a thorough treatment of MDPs, see Bertsekas [3] and Puterman [27].

An MDP is composed of a finite set of *states*. One of the states is designated as the *initial* state. In each time unit, the *controller* of the MDP has to choose one

of several *actions* associated with the current state of the process. Each action has an immediate *cost* (or *reward*) and a probability distribution over the next state of the process associated with it. In a *finite horizon* MDP, the controller has to guide the process for a given number of steps. In an *infinite horizon* MDP, the controller has to guide the process indefinitely. The goal of the controller is to minimize the overall cost associated with the actions chosen. In finite horizon problems, the overall cost may be taken to be the sum of the immediate costs of all the actions performed. In infinite horizon problems, the overall cost is either taken to be the limit of the *average cost per time unit*, or the total *discounted* cost, which is defined as the sum of all immediate costs, with the cost of the i -th action is multiplied by λ^i , where $0 < \lambda < 1$ is a given *discount factor*. It is possible to interpret the discount factor λ as either a rate of *deflation*, or as the probability in which the process stops at each stage. All MDPs considered in this paper are infinite horizon discounted MDPs.

A *policy* (or *strategy*) for an MDP is a rule that specifies which action should be taken in each situation. The decision may depend on the current state of the process and possibly on the *history*. A policy is said to be *positional* if it is deterministic and history independent. A policy is said to be *optimal* if it minimizes the overall cost. One of the fundamental results concerning MDPs, see, e.g., [27], says that every infinite horizon MDP has an optimal positional policy. Furthermore, there is always a single positional policy which is optimal for every starting state. By *solving* an MDP, we mean finding such an optimal strategy, and the optimal overall cost for each starting state.

d'Epenoux [8] seems to be the first to observe that MDPs can be solved using *linear programming*. As linear programs can be solved in polynomial time (Khachiyan [18], Karmarkar [16]), we get that MDPs, under both the limiting average cost and the discounted cost criteria, can also be solved in polynomial time.

The known polynomial time algorithms for linear programming, and hence the derived algorithms for solving MDPs, are not *strongly polynomial*. It is a major open problem whether there are strongly polynomial

*SRI International, AI Center, 333 Ravenswood Ave., Menlo Park, CA 94025, USA. E-mail: madani@ai.sri.com

†AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932, USA. E-mail: mthorup@research.att.com

‡School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: zwick@cs.tau.ac.il. Research supported by BSF grant no. 2006261.

time algorithms for solving MDPs. Ye [30] obtained a strongly polynomial time interior-point algorithm for solving discounted MDPs when the discount factor $0 \leq \lambda < 1$ is a fixed constant.

MDPs are often solved in practice using a method called *policy iteration* (see, e.g., [3],[27]). Although this method seems to work very well, it is not known whether its worst-case running time is polynomial. Analyzing the various variants of the policy iteration algorithm and determining whether any of them run in worst-case polynomial time is a major challenge for the theoretical computer science community. (See also [25],[24],[22].)

An MDP is said to be *deterministic* (or *non-stochastic*) if each action uniquely determines the next state of the process. In other words, the probability distribution associated with each action assigns probability 1 to one of the states. Deterministic MDPs (DMDPs) form an interesting subclass of MDPs. A DMDP can be conveniently represented as a weighted directed graph. The vertices of the graph correspond to the states of the DMDP and the edges correspond to the action. The weight of an edge is the immediate cost of the corresponding action.

One of the motivations for studying DMDPs is that they arise in the solution of Mean Payoff Games (MPGs) or Discounted Payoff Games (DPGs) (see [10],[12],[33]). Similarly, general MDPs arise in the solution of Simple Stochastic Games (SSGs) (see [6]). SSGs, MPGs and DPGs may be viewed as competitive versions of MDPs and DMDPs in which the single controller is replaced by two adversarial players. It is a major open problem whether SSGs, MPGs and DPGs can be solved in polynomial time. The fastest known algorithms are randomized and sub-exponential (see [20],[4],[13]).

Solving DMDPs with the limiting average cost criteria is equivalent to well known problem of finding a *minimum mean weight cycle* in a directed graph. A strongly polynomial $O(mn)$ -time algorithm for this problem was given by Karp [17]. (See also Madani [23].) (Here n is the number of vertices and m is the number of edges.) An $O(mn + n^2 \log n)$ -time algorithm, which performs very well in practice, was obtained by Young, Tarjan and Orlin [31].

Solving discounted DMDPs, i.e., DMDPs with the discounted cost criteria, seems to be a somewhat harder problem. Strongly polynomial time algorithms for discounted DMDPs were obtained by Papadimitriou and Tsitsiklis [26], Madani [21], and most recently by Andersson and Vorobyov [1]. The algorithm of Andersson and Vorobyov [1], the fastest of these algorithms, runs in $O(mn^2)$ time.

We present two new algorithms for solving discounted DMDPs. (The initial submission contained

only the first of these algorithms.) The first one runs in $O(mn + n^2 \log n)$ -time, while the second runs in $O(mn)$ time. The first algorithm has the advantage that its running time is in many cases much better than its worst running time, while the second algorithm always runs in $\Theta(mn)$ time. The two algorithms may be viewed as adaptations of algorithms by Young, Tarjan and Orlin [31] and Karp [17] for the finding minimum mean-weight cycles.

We also consider algorithms of the Discounted All-Pairs Shortest Paths (DAPSP) problem. Namely, given a weighed directed graph $G = (V, E)$ and given a discount factor $0 < \lambda < 1$, find for every pair of vertices u and v a path from u to v of smallest discounted cost. Most techniques used to solve standard, i.e., non-discounted shortest paths problem do *not* work in the discounted setting as a subpath of a shortest discounted path is *not* necessarily a shortest discounted path. (This is discussed further in Section 5.) Nevertheless, by combining several different techniques we obtain an $\tilde{O}(m^{1/2}n^2)$ algorithm for the DAPSP problem, essentially matching, for dense graphs, the time bounds known for the standard APSP problem. This improves an $O(n^4)$ -time (parallelizable) algorithm for the problem given by Papadimitriou and Tsitsiklis [26].

2 Deterministic Markov decision processes

A Deterministic Markov Decision Process (DMDP) is a weighted directed graph $G = (V, E, c)$, where V is a set of *states*, or *vertices*, $E \subseteq V \times V$ is a set of *actions*, or directed *edges*, and $c : E \rightarrow \mathbb{R}$ is a *cost* function. We assume that each vertex has at least one outgoing edge.

The *controller* of a DMDP G selects an infinite path $P = \langle v_0, v_1, \dots \rangle$, that starts at given start vertex v_0 . For a given *discount factor* λ , where $0 < \lambda < 1$, the λ -*discounted* cost of the infinite path P is defined as:

$$c_\lambda(P) = \sum_{i=0}^{\infty} \lambda^i c(v_i, v_{i+1}).$$

The goal of the controller is to find, from every start vertex, an infinite path whose discounted cost is as small as possible. The *value* $x_\lambda(u)$ of a vertex u of G is defined accordingly as:

$$(1) \quad x_\lambda(u) = \inf \left\{ c_\lambda(P) \mid \begin{array}{l} P \text{ is an infinite path} \\ \text{in } G \text{ that starts at } u \end{array} \right\}.$$

We show below that this infimum is always attained by an ultimately periodic infinite path P . We usually consider a fixed discount factor λ and write $c(P)$ and $x(u)$ instead of $c_\lambda(P)$ and $x_\lambda(u)$.

A *strategy* for a DMDP $G = (V, E, c)$ is a function $\sigma : V \rightarrow V$ such that for every $u \in V$ we have

$(u, \sigma(u)) \in E$. In other words, a strategy corresponds to the selection of one outgoing edge from each vertex. Given an initial vertex v_0 , a strategy σ defines an infinite path $Path(\sigma, v_0) = \langle v_0, v_1, \dots \rangle$, where $v_{i+1} = \sigma(v_i)$, for every $i \geq 0$. Note that $Path(\sigma, v_0)$ is composed of a (possibly empty) initial path that leads to a cycle which is repeated over and over again.

Strategies, as defined above are *positional* (i.e., *memoryless*) and *pure* (i.e., *deterministic*). It is possible to define a much wider class of strategies in which the edge chosen at each stage depends on the initial path already formed. Furthermore, the choice could be probabilistic. It is not difficult to show, as we do below, that for any given discount factor λ , the controller of a DMDP always has a single positional strategy that produces optimal paths from every starting vertex. We present two improved algorithm for finding such an optimal strategy.

The following well known Theorem characterizes optimal values and strategies.

THEOREM 2.1. *Let $G = (V, E, c)$ be a DMDP and let $0 < \lambda < 1$ be a discount factor. Then, the values $x(u) = x_\lambda(u)$ of the vertices of G are the unique solution of the following set of equations:*

$$(2) \quad x(u) = \min_{v: (u,v) \in E} \lambda x(v) + c(u, v), \quad \forall u \in V.$$

Furthermore, if for every $u \in E$ we let $\sigma(u)$ be a vertex for which $(u, \sigma(u)) \in E$ and $x(u) = \lambda x(\sigma(u)) + c(u, v)$, then σ is an optimal strategy, i.e., $c(Path(\sigma, u)) = x(u)$, for every $u \in V$.

The equations in Theorem 2.1 are sometimes called the *Bellman equations*. As a corollary of Theorem 2.1, we also get:

COROLLARY 2.1. *Let $G = (V, E, c)$ be a DMDP and let $0 < \lambda < 1$ be a discount factor. Then, the values $x(u) = x_\lambda(u)$ of the vertices of G are the unique optimal solution of the following linear program:*

$$(3) \quad \begin{array}{ll} \max & \sum_{u \in V} x(u) \\ \text{s.t.} & x(u) \leq \lambda x(v) + c(u, v), \quad \forall (u, v) \in E \end{array}$$

The linear program (3) corresponding to a DMDP has a very special form. First, it contains at most *two variables per inequality*. (If G contains self loops, then the corresponding inequalities have only one variable appearing in them.) Second, in each inequality, when written in the conventional form $x(u) - \lambda x(v) \leq c(u, v)$, the coefficient of one variable is positive while the coefficient of the other coefficient is negative. Strongly polynomial time algorithms for checking the feasibility

of linear programs with at most two variables per inequality were obtained by Cohen and Megiddo [5] and Hochbaum and Naor [14]. Madani [21] utilizes these results to obtain an $O(mn^2 \log n)$ -time algorithm for solving discounted DMDPs. His algorithm actually solves a slightly more general problem in which each edge may have a different discount factor associated with it. Andersson and Vorobyov [1] recently obtained an $O(mn^2)$ -time algorithm for solving a DMDP with a single discount factor associated with all edges. In the next section we describe an improved version of their algorithm.

3 First algorithm for discounted DMDP

We now describe our $O(mn + n^2 \log n)$ -time algorithm for solving discounted DMDPs. For simplicity, we assume that $c(u, v) \geq 0$, for every $(u, v) \in E$. If this condition is not satisfied, we add $\alpha = -\min_{(u,v) \in E} c(u, v)$ to all edge costs. This increases the values of all vertices by exactly $\alpha \sum_{i=0}^{\infty} \lambda^i = \alpha/(1 - \lambda)$.

The basic idea. The algorithm maintains for each vertex u a number $val[u]$. The values always satisfy the inequalities

$$(4) \quad val[u] \leq \lambda val[v] + c(u, v), \quad \forall (u, v) \in E.$$

Initially we set $val[u] = 0$ for every $u \in V$. The inequalities are then satisfied as $c(u, v) \geq 0$, for every $(u, v) \in E$. An edge $(u, v) \in E$ is said to be *tight* if $val[u] = \lambda val[v] + c(u, v)$. The algorithm gradually increases these values at varying speeds, making sure that selected tight edges remain tight, until each vertex $u \in V$ has an outgoing tight edge. By Theorem 2.1, the values obtained are the required solution.

For each vertex $u \in V$, we let $parent[u]$ be a vertex v for which the edge $(u, v) \in E$ is tight, if such a vertex exists. Otherwise, $parent[u] = nil$. The parent pointers define a *pseudo-forest*, i.e., a directed graph in which the outdegree of each vertex is at most 1. Each connected component in a pseudo-forest is either a directed *tree*, in which all edges are directed towards a root, or a *pseudo-tree* in which directed trees are attached to a unique cycle. When $parent[u] \neq nil$ for every $u \in V$, we are done. A small pseudo-forest, containing one pseudo-tree and two trees is shown in Figure 1. (The meaning of the dotted edges and shaded vertices is explained later.) Note that a tree, according to our definition, is *not* a pseudo-tree, while a forest *is* a pseudo-forest.

If vertex $u \in V$ is contained in a tree, we let $depth[u]$ be the *depth* of u in its tree. The depth of a root is 0. In an attempt to obtain tight outgoing edges for the roots of the remaining trees, we gradually increase the values of all vertices contained in trees. Values of vertices that belong to pseudo-trees remain

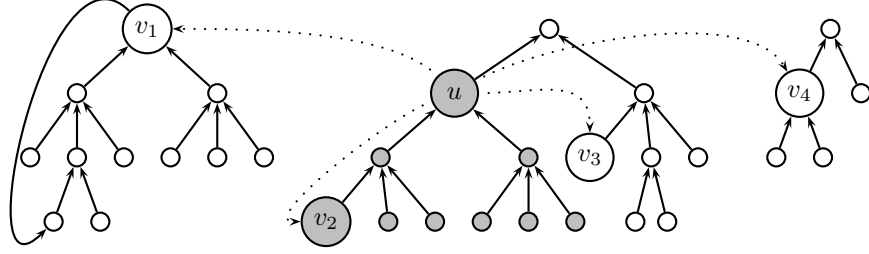


Figure 1: A simple pseudo-forest composed of one pseudo-tree and two trees.

unchanged. It is easy to check that if, for some $t \geq 0$, we simultaneously increase the value of each vertex u currently contained in a tree by $\lambda^{\text{depth}[u]}t$, then all tree edges remain tight. We let $\text{speed}[u] = \lambda^{\text{depth}[u]}$, if u is contained in a tree, and $\text{speed}[u] = 0$, otherwise.

How large can t be so that no edge constraint is violated? Let $(u, v) \in E$. If $\text{speed}[u] > \lambda \text{speed}[v]$, then the inequality

$$\text{val}[u] + \text{speed}[u]t \leq \lambda(\text{val}[v] + \text{speed}[v]t) + c(u, v)$$

holds only for $t \leq \text{time}(u, v)$, where

$$\text{time}(u, v) = \frac{-\text{val}[u] + \lambda \text{val}[v] + c(u, v)}{\text{speed}[u] - \lambda \text{speed}[v]}.$$

Note that $\text{speed}[u] > \lambda \text{speed}[v]$ if and only if $\text{depth}[u] \leq \text{depth}[v]$. If $\text{speed}[u] \leq \lambda \text{speed}[v]$, then the required inequality holds for any $t \geq 0$, so we let $\text{time}(u, v) = +\infty$. Clearly, $\text{time}(u, v) \geq 0$, for every $(u, v) \in E$. If (u, v) is a tree edge, then $\text{speed}[u] = \lambda \text{speed}[v]$ and hence $\text{time}(u, v) = +\infty$. If $(u, v) \in E$ and u is a root, then as roots have the highest speed we have $\text{speed}[u] > \lambda \text{speed}[v]$ and thus $\text{time}(u, v) < +\infty$. Let

$$t^* = \min_{(u, v) \in E} \text{time}(u, v).$$

As each root has at least one edge leaving it, we get that $0 \leq t^* < +\infty$. (We may have $t^* = 0$.) At time t^* , all inequalities still hold, and at least one edge $(u, v) \in E$ that does not belong to the pseudo-forest is tight. We let v be the new parent of u , by letting $\text{parent}[u] \leftarrow v$. If there are still vertices without tight outgoing edge we repeat the process.

What happens to the pseudo-forest as a result of the assignment $\text{parent}[u] \leftarrow v$? The four dotted edges emanating from u in Figure 1 describe the four possible cases. If v , like v_1 in the figure, is contained in a pseudo-tree, then u and all its descendants, shown in light gray in Figure 1, become part of this pseudo-forest. If v , like v_2 , is a descendant of u , then the descendants of u form a new pseudo-tree. If v , like v_3 , belongs to the same tree as u , but is not a descendant of u , then u

and its descendants change their position in the tree. Finally, if v , like v_4 in the figure, belongs to a different tree, then u and its descendants move to the tree of v . We show below that in the last two cases we must have $\text{depth}[v] \geq \text{depth}[u]$.

How do we know that the above process would stop, and after how many iterations? Let $(u, v) \in E$ be the tight edge identified by the process above. If u is a root, then we have definitely made progress, as the number of trees decreased by 1. Unfortunately, this may not happen very often. We show, however, that we do make some progress even if u is not a root. As $\text{time}(u, v) < +\infty$, we get that $\text{speed}[u] > \lambda \text{speed}[v]$. This, in turn, implies that either $\text{speed}[v] = 0$, i.e., v is contained in a pseudo-tree, or that v is contained in a tree and $\text{depth}[v] \geq \text{depth}[u]$. (Note that this holds even if u and v are in different trees.) Thus, making v the new parent of u increases the depth of u , and of all its descendants. The depth of all other vertices remains unchanged. As there are n vertices and the depth of each one of them cannot exceed n , we get that the number of iterations performed is at most n^2 .

The algorithm described above is similar to the algorithm suggested by Andersson and Vorobyov [1]. They, however, only increase the values of the vertices contained in the smallest tree of the pseudo-forest. We find it more natural to increase all values simultaneously. This also simplifies the implementation.

Naive implementation. In each iteration compute $\text{time}(u, v)$, for every edge $(u, v) \in E$. Compute $t^* = \min_{(u, v) \in E} \text{time}(u, v)$ and let $(u, v) \in E$ be such that $\text{time}(u, v) = t^*$. Let $\text{val}[w] \leftarrow \text{val}[w] + \text{speed}[w]t^*$, for every $w \in V$. Next, update $\text{speed}[w]$ for every descendant w of u . More specifically, if v belongs to a pseudo-tree, or if v is a descendant of u , in which case a new pseudo-tree is formed, let $\text{speed}[w] \leftarrow 0$, for every descendant w of v . Otherwise, let $\text{fact} \leftarrow \lambda \text{speed}[v] / \text{speed}[u]$, and then $\text{speed}[w] \leftarrow \text{fact} \cdot \text{speed}[w]$, for every descendant w of v . To access the descendant of a given vertex we maintain, for every vertex, a list of its children. Finally, let $\text{parent}[u] \leftarrow v$.

Each iteration can easily be implemented in $O(m)$ time. As there are at most n^2 iterations, the total running time of the algorithm is $O(mn^2)$, matching the running time of the algorithm of Andersson and Vorobyov [1]. We next describe two more efficient implementations of the algorithm.

Efficient implementation. We do not ‘reset the clock’ after each iteration. We adopt the convention that $val[u]$ is the value of vertex u at time 0. The actual value of u at time t is $val[u] + speed[u] \cdot t$. If at time t , the speed in which $val[u]$ is increased is changed from $speed[u]$ to $speed'[u]$, we let

$$val[u] \leftarrow val[u] + (speed[u] - speed'[u])t.$$

If we denote by $value'[u]$ the new value of u , we get that

$$value'[u] + speed'[u] \cdot t = val[u] + speed[u] \cdot t.$$

We can thus behave as if the initial value of u was $value'[u]$ and it was increased from time 0 to time t at a speed of $speed'[u]$. Note also that when $speed'[u] = 0$, as happens when the speed of u is changed for the last time, $val[u]$ gets the value of u at time t , which then remains unchanged. The advantage of this setting is that if $(u, v) \in E$, and $speed[u]$ and $speed[v]$ do not change in an iteration, then neither does $time(u, v)$. If $speed[u]$ or $speed[v]$ change, then $time(u, v)$ may increase or decrease, but as the inequality corresponding to (u, v) holds at time t , we would still have $time(u, v) \geq t$.

We maintain a *priority queue* that holds all the edges of the graph. The key associated with an edge $(u, v) \in E$ is $time(u, v)$. In each iteration we use a **find-min** operation to find an edge (u, v) with the smallest $time(u, v)$. We let $T = T_u$ be the set of descendants of u in the current tree. We update the speeds of all vertices in T , as done in the naive implementation, and let $parent[u] \leftarrow v$. Next, for every edge $(u', v') \in E$ where $u' \in T$ or $v' \in T$, we recompute $time(u', v')$ and update the key of (u', v') in the priority queue. (We note that $time(u', v')$ may either increase, remain unchanged, or decrease.) The key of (u', v') is updated by first deleting (u', v') from the priority queue and then reinserting it with its new key. If $u', v' \notin T$, then $time(u', v')$ is unchanged.

Note that in each iteration the algorithm changes only one pseudo-forest edge. It may happen that at some stage two (or more) edges (u_1, v_1) and (u_2, v_2) have the smallest time associated with them, i.e., $time(u_1, v_1) = time(u_2, v_2) = \min_{(u,v) \in E} time(u, v)$. The algorithm arbitrarily chooses one of these edges, say (u_1, v_1) , and modifies the pseudo-forest accordingly. The second edge (u_2, v_2) is then handled in one of the next iterations. Thus, two changes may have the same

‘time’ associated with them, but they are handled sequentially by the algorithm.

As argued above, the depth of all the vertices in T_u is increased. (To make this claim hold in all cases, we define the depth of vertices contained in pseudo-trees to be n .) Thus, for every edge $(u', v') \in E$, $time(u', v')$ is computed at most $2n$ times. Overall, we perform at most $O(mn)$ **insert** and **delete** operations, and at most $O(n^2)$ **find-min** operations. As each one of these operation takes $O(\log n)$ time, using a simple binary heap (see, e.g., [7]), the total running time of the algorithm is $O(mn \log n)$.

More efficient implementation Several priority queue implementations, such as Fibonacci heaps [11], support **decrease-key** operations in constant *amortized* time. Unfortunately, such priority queues cannot be used directly to speed-up the implementation of the algorithm described above as $time(u', v')$ may increase, as well as decrease.

To take advantage of constant time decrease-key operations, we maintain a priority queue that holds vertices instead of edges. For every vertex $u \in V$, we let $(u, next[u])$ be an outgoing edge of u such that $time(u, next[u]) \leq time(u, v)$, for every $(u, v) \in E$. We also let $time[u] = time(u, next[u])$. All vertices still contained in trees are now maintained in a priority queue PQ . The key of a vertex $u \in V$ is $time[u]$.

While there are still vertices with no tight outgoing edges, i.e., while there are still vertices contained in trees, or equivalently $PQ \neq \phi$, we use a **find-min**(PQ) operation to find a vertex u with minimal value of $time[u] = time(u, next[u])$. As before, we let $T = T_u$ be the subtree of u and update the speeds of all vertices in T . This takes $O(|T|)$ time. The only edges that we need to reexamine now are edges of the form (w, v) or (v, w) , where $w \in T$ and $v \in V$.

We first handle edges that emanate from T . If T joins an existing pseudo-forest (i.e., $next[u] \notin PQ$), or becomes a new pseudo-forest (i.e., $next[u] \in T$), we remove all the vertices of T from PQ and there is no need to reexamine their outgoing edges.

If T remains part of a tree (i.e., $next[u] \in PQ - T$), we temporarily **delete** each vertex $w \in T$ from PQ , recompute $next[w]$ and $time[w]$ by examining all its outgoing edges, and then re-**insert** w to PQ with its new key.

In both cases, we still need to reexamine edges that enter T . Let $(v, w) \in E$, where $v \in PQ$ and $w \in T$, be such an edge. If $v \in T$, then the edge (v, w) was already handled. Assume, therefore, that $v \notin T$. As $speed[v]$ remained unchanged while $speed[w]$ decreased, $time(v, w)$ can only decrease. Thus, if $time(v, w) < time[v]$, we can simply set $next[v] \leftarrow w$, $time[v] \leftarrow time(v, w)$, and

<hr/> Function DetMDP1 ($G = (V, E, c), \lambda$) <hr/> <pre> PQ ← ϕ foreach u ∈ V do parent[u] ← nil val[u] ← 0 speed[u] ← 1 foreach u ∈ V do scan(u) while PQ ≠ ϕ do u ← find-min(PQ) /* Find next tight edge */ parent[u] ← next[u] T ← subtree(u) if parent[u] ∈ T then fact ← 0; /* New cycle formed */ else fact ← λ · speed[parent[u]]/speed[u] foreach v ∈ T do /* Update subtree vertices */ delete(PQ, v) val[v] ← val[v] + (1 - fact) · speed[v] · time[u] speed[v] ← fact · speed[v] foreach v ∈ T do if speed[v] > 0 then scan(v) /* Handle outgoing edges */ foreach (w, v) ∈ E do examine(w, v); /* Incoming edges */ </pre> <hr/>	<hr/> Function scan (u) <hr/> <pre> next[u] ← nil; time[u] ← +∞; insert(PQ, u, time[u]); foreach (u, v) ∈ E do examine(u, v) </pre> <hr/> Function examine (u, v) <hr/> <pre> if time(u, v) < time[u] then next[u] ← v; time[u] ← time(u, v); decrease-key(PQ, u, time[u]) </pre> <hr/> Function time (u, v) <hr/> <pre> if speed[u] > λ · speed[v] then return (-val[u] + λ · val[v] + c(u, v))/(speed[u] - λ · speed[v]) else return +∞ </pre> <hr/>
---	--

Figure 2: Algorithm DetMDP1 for solving deterministic Markov Decision Processes.

use a **decrease-key** operation to decrease the key of v in PQ .

Overall, the algorithm performs $O(mn)$ **decrease-key** operations and $O(n^2)$ **insert**, **delete** and **find-min** operations. As the amortized cost of each **decrease-key**, **find-min** and **insert** operation is $O(1)$, while the amortized cost of a **delete** operation is $O(\log n)$, the total running time of the algorithm is $O(mn + n^2 \log n)$.

A complete implementation of the algorithm, which we call DetMDP1, is given in Figure 2. The only operation not explicitly shown in Figure 2 is the maintenance of children lists. Algorithm DetMDP1 uses a call $T \leftarrow \text{subtree}(u)$ to obtain the set of descendants of u . To implement this operation in $O(|T|)$ time, we need to maintain for each vertex u a list $children[u]$ of its children. The maintenance of these lists is simple. Before setting $parent[u] \leftarrow v$, we delete u from $children[parent[u]]$, if $parent[u] \neq nil$, and then insert u to $children[v]$. Algorithm DetMDP1 uses a function $\text{time}(u, v)$ that computes the time in which an edge (u, v) becomes tight. It also uses a function $\text{examine}(u, v)$ that recomputes $\text{time}(u, v)$ and updates $\text{time}[u]$ and $\text{next}[u]$ if necessary. If $\text{time}[u]$ is decreased, then it uses a **decrease-key** operation to decrease the key of u in PQ to the new

value of $\text{time}[u]$. Finally, $\text{scan}(u)$ initializes $\text{time}[u]$ and $\text{next}[u]$ to $+\infty$ and nil , respectively, and then examines all the outgoing edges of u .

We summarize the result obtained in this section in the following theorem:

THEOREM 3.1. *Algorithm DetMDP1 solves discounted DMDPs in $O(mn + n^2 \log n)$ time, where n is the number of vertices and m is the number of edges.*

4 Second algorithm for discounted DMDP

An adaptation of Karp's algorithm for solving discounted DMDPs is given in Figure 3. The algorithm starts by computing $d_k(u)$, the minimal discounted cost of a k -edge path that starts at u , for every $u \in V$ and every $0 \leq k \leq n$. This is easily done on $O(mn)$ time using a simple Bellman-Ford stage. The algorithm then computes

$$y_0(v) = \max_{0 \leq k < n} \frac{d_n(v) - \lambda^{n-k} d_k(v)}{1 - \lambda^{n-k}}, \forall v \in V.$$

These ratios $(d_n(v) - \lambda^{n-k}) / (1 - \lambda^{n-k})$ are the analogues of the non-discounted ratios $(d_n(v) - d_k(v)) / (n - k)$ that appear in Karp's original algorithm. In the non-

Function DetMDP2($G = (V, E, c), \lambda$)

```

foreach  $u \in V$  do
   $d_0(u) \leftarrow 0$ 
for  $k \leftarrow 1$  to  $n$  do
  foreach  $u \in V$  do
     $d_k(u) = \min_{v:(u,v) \in E} c(u,v) + \lambda d_{k-1}(v)$ 
foreach  $v \in V$  do
   $y_0(v) \leftarrow \max_{0 \leq k < n} \frac{d_n(v) - \lambda^{n-k} d_k(v)}{1 - \lambda^{n-k}}$ 
for  $k \leftarrow 1$  to  $n-1$  do
  foreach  $u \in V$  do
     $y_k(u) = \min_{v:(u,v) \in E} c(u,v) + \lambda y_{k-1}(v)$ 
foreach  $v \in V$  do
   $x(v) \leftarrow \min_{0 \leq k < n} y_k(v)$ 

```

Figure 3: A discounted version of Karp’s algorithm.

discounted case, the expression

$$\mu^* = \min_{v \in V} \max_{0 \leq k < n} \frac{d_n(v) - d_k(v)}{n - k}$$

is the minimum mean weight of a cycle in the graph. In the discounted case, things are more complicated. In particular, we are after n potentially different values, and not just one. We show below that $x(v) \leq y_0(v)$, for every $v \in V$, and that on every optimal cycle there is at least one vertex v for which $x(v) = y_0(v)$. To compute the correct values of all vertices algorithm DetMDP2 performs a second Bellman-Ford stage in which the $y_0(v)$ values serve as the initial values of the vertices. For every $v \in V$ and $0 \leq k < n$, $y_k(v)$ is then the minimal discounted cost of a k -edge path that starts at v , where the discount cost now takes into account the initial value of the last vertex on the path. Finally, the algorithm computes $x(v) = \min_{0 \leq k < n} y_k(v)$, for every $v \in V$. The running time of the whole algorithm is clearly $O(mn)$. Correctness follows from the following theorem:

THEOREM 4.1. (i) For every vertex $v \in V$ we have $x(v) \leq y_0(v)$. (ii) On every optimal cycle in G there is at least one vertex v for which $x(v) = y_0(v)$.

To facilitate the proof of Theorem 4.1, we add to G an auxiliary sink s and add zero cost edges from all vertices of G to s . We let $\bar{G} = (\bar{V}, \bar{E})$ denote this extended graph. Any k -edge path $P = \langle u = v_0, v_1, \dots, v_k \rangle$ starting at a vertex u in G can be extended to a $(k+1)$ -edge path $\bar{P} = \langle u = v_0, v_1, \dots, v_k, s \rangle$ from u to s in \bar{G} . As $c(v_k, s) = 0$, the discounted cost of these two paths is the same.

For every *potential* function $p : \bar{V} \rightarrow \mathbb{R}$, we define modified edge costs as follows:

$$c'(u, v) = c'_p(u, v) = c(u, v) - p(u) + \lambda p(v), \quad (u, v) \in \bar{E}.$$

If $P = \langle v_0, v_1, \dots, v_k \rangle$ is a path in \bar{G} , then

$$c'(P) = c(P) - p(u) + \lambda^k p(v).$$

We refer to $c(P)$ as the c -cost of P and to $c'(P)$ as the c' -cost of P . For every potential function p , if P is a k -edge path starting at u with minimal c -cost in G , then \bar{P} is a $(k+1)$ -edge path from u to s with minimal c' -cost in \bar{G} .

Let $d'_k(u)$ be the minimal discounted c' -cost of a $(k+1)$ -edge path from u to s in \bar{G} . Clearly:

$$d_k(u) = d'_k(u) + p(u) - \lambda^{k+1} p(s).$$

In the proof of Theorem 4.1 we use the following useful potential function $p(u) = x(u)$, for $u \in V$. The value $p(s)$ will be set in a way to be explained below. (Note that this potential function is only used in the proof. It is not used by the algorithm.) With this potential function, we have $c'(u, v) \geq 0$, for every $(u, v) \in E$, and $c'(u, v) = 0$ if and only if $(u, v) \in E$ is an optimal edge. The only edges that can have negative modified costs are edges of $\bar{E} - E$, i.e., edges to the new sink t .

Proof. (Of Theorem 4.1(i).) We have to show that for every vertex $v \in V$ there exists $0 \leq k < n$, such that

$$d_n(v) - \lambda^{n-k} d_k(v) \geq (1 - \lambda^{n-k}) x(v).$$

For every $u \in V$ and $0 \leq k < n$ we have

$$\begin{aligned} & d_n(v) - \lambda^{n-k} d_k(v) = \\ & (d'_n(v) + x(v) - \lambda^{n+1} p(s)) - \lambda^{n-k} (d'_k(v) + x(v) - \lambda^{k+1} p(s)) \\ & = (d'_n(v) - \lambda^{n-k} d'_k(v)) + (1 - \lambda^{n-k}) x(v). \end{aligned}$$

(Note that this holds for *any* choice of $p(s)$, the potential of the auxiliary sink.) It is enough to show, therefore, that for every $v \in V$ there exists $0 \leq k < n$ such that $d'_n(v) \geq \lambda^{n-k} d'_k(v)$.

Consider an n -edge path P that attains the value $d'_n(v)$. The path \bar{P} then attains the value $d'_n(v)$. As P is composed of n edges, it must contain a repeated vertex. We thus have $P = P_1 P_2 P_3$, where P_2 is a cycle. (This representation is not necessarily unique.) Let $\ell_1, \ell_2, \ell_3 \geq 0$ be the number of edges on P_1, P_2 and P_3 . Note that $\ell_2 > 0$. Let $P' = P_1 P_3$ be the path obtained from P by removing the cycle P_2 . For every edge (u, v) on P we have $c'(u, v) \geq 0$. Hence, $c'(P_1), c'(P_2) \geq 0$. Thus

$$\begin{aligned} d'_n(v) &= c'(P_1) + \lambda^{\ell_1} c'(P_2) + \lambda^{\ell_1 + \ell_2} c'(P_3) \\ &\geq \lambda^{\ell_2} (c'(P_1) + \lambda^{\ell_1} c'(P_3)) \geq \lambda^{\ell_2} d'_{n-\ell_2}(v). \end{aligned}$$

For $k = n - \ell_2$ we thus have $d'_n(v) \geq \lambda^{n-k} d'_k(n)$, as required. \square

Proof. (Of Theorem 4.1(ii).) As we showed in the proof of Theorem 4.1(i), for every $v \in V$ and every $0 \leq k < n$ we have

$$d_n(v) - \lambda^{n-k} d_k(v) = (d'_n(v) - \lambda^{n-k} d'_k(v)) + (1 - \lambda^{n-k}) x(v).$$

It is thus enough to show that on every optimal cycle C there is at least one vertex v for which $d'_n(v) \leq \lambda^{n-k} d'_k(v)$, for every $0 \leq k < n$. Note that we are still free to choose $p(s)$, the potential assigned to the sink s . (The values $d'_k(v)$ depend on $p(s)$, but $p(s)$ cancels out in $d'_n(v) - \lambda^{n-k} d'_k(v)$.)

Let C be an optimal cycle. For every edge (u, v) on C we have $c'(u, v) = 0$. Let u be an arbitrary vertex on C . Choose $p(s)$ such that $\min_{0 \leq k < n} d'_k(u) = 0$. (If $d'_k(u)$ are the values obtained with $p(s) = 0$, let $p(s) = \max_{0 \leq k < n} -d'_k(v) / \lambda^{k+1}$.) We still have $c'(u, v) \geq 0$, for every $(u, v) \in E$.

Suppose now that $d'_\ell(u) = 0$ and let P be an ℓ -edge path starting at u such that $c'(\bar{P}) = 0$. Extend P to an n -edge path by adding to its beginning edges from the optimal cycle C . Let P' be the n -edge path formed and let v be its starting point. As all the edges added to P have a c' -cost of 0, we clearly have $c'(\bar{P}') = 0$ and thus $d'_n(v) \leq 0$. We claim that for every $0 \leq k < n$ we have $d'_k(v) \geq 0$, and thus $d'_n(v) \leq 0 \leq \lambda^{n-k} d'_k(v)$, for every $0 \leq k < n$, as required.

It remains to show, therefore, that $d'_k(v) \geq 0$, for every $0 \leq k < n$. Suppose, for the sake of contradiction, that $d'_k(v) < 0$, for some $0 \leq k < n$. Let Q be a k -edge path starting at v such that $c'(\bar{Q}) < 0$. Extend Q , by adding to its beginning edges from C , to a path Q' that start at u , the vertex chosen above. (Recall that u and v are both on C .) As all edges added have a c' -cost of 0, we have $d'(Q') < 0$. If Q' contains n or more edges, then it contains a cycle. By removing such a cycle we get a shorter path Q'' that starts at u . As the c' -cost of the cycle is non-negative, we get that $c'(Q'') < 0$. By repeatedly removing such cycles, we get a path Q'' that starts at u that contains less than n edges for which $c'(Q'') < 0$, a contradiction to the assumption that $\min_{0 \leq k < n} d'_k(u) = 0$. \square

5 Discounted All-Pairs Shortest Paths

Let $G = (V, E, c)$ be a weighted directed graph, and let $0 < \lambda < 1$ be a discount factor. The discounted cost of a path $P = \langle v_0, v_1, \dots, v_k \rangle$, is defined, as before, as $c(P) = \sum_{i=0}^{k-1} \lambda^i c(v_i, v_{i+1})$.

DEFINITION 5.1. (DISCOUNTED SHORTEST PATHS)

Let $u, v \in V$ and let $k \geq 0$. We let $d_k(u, v)$ the

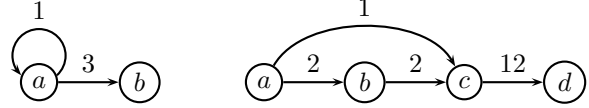


Figure 4: Two simple graphs.

minimum discounted cost of a path from u to v that contains exactly k edges. (If no such path exists, then $d_k(u, v) = +\infty$.) A k -edge path P from u to v for which $c(P) = d_k(u, v)$ is said to be a k -edge discounted shortest path from u to v . We let $d(u, v) = \inf_k d_k(u, v)$ be the discounted distance from u to v . A path P from u to v for which $c(P) = d(u, v)$ is said to be a discounted shortest path from u to v .

We note that the infimum in the definition of $d(u, v)$ may not be attained by a finite path, even if all edge costs are positive. Consider the graph on the lefthand side of Figure 4. If $\lambda = \frac{1}{2}$, then $d(a, b) = 1 + \frac{1}{2} + \frac{1}{4} + \dots = 2$. Also note that a *prefix* of discounted shortest path is not necessarily a discounted shortest path. Consider for example the graph on the righthand side of Figure 4, again with $\lambda = \frac{1}{2}$. The shortest discounted path from a to d is $a \rightarrow b \rightarrow c \rightarrow d$ whose discounted cost is $2 + \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 12 = 6$. (Note that the discounted cost of the path $a \rightarrow c \rightarrow d$ is $1 + \frac{1}{2} \cdot 12 = 7$.) The prefix $a \rightarrow b \rightarrow c$, whose discounted cost is $2 + \frac{1}{2} \cdot 2 = 3$, is *not* a discounted shortest path from a to c , as the discounted cost of the path $a \rightarrow c$ is 1.

Due to these, and other reasons, simple adaptations of classical algorithms for solving shortest paths problems *cannot* be used to find discounted shortest paths. Both the forward and backward versions of Dijkstra's algorithm, for example, fail, even if all edge costs are positive. Similarly, the simple matrix-multiplication algorithm and the Floyd-Warshall algorithms also do not work correctly. (Finding out where each one of these algorithms fail is an easy but interesting exercise.)¹

The only algorithm that does seem to remain applicable is the robust, but slow, Bellman-Ford algorithm. Using it, however, to solve the All-Pairs version of the problem requires $O(mn^2)$ time.

It is still true, however, that a *suffix* of a discounted shortest path is a discounted shortest path, and that prefix of length k of a discounted shortest path is a k -edge discounted shortest. It is also not difficult to see that $d(u, v)$ is either attained by a simple finite path, or can be approached by a collection of paths of the form PC^iQ , $i \geq 0$, where P is a simple path from u to some

¹Littman [19] attempts to cast discounted DMDPs and discounted shortest paths problems as *closed semiring* problems. His attempt, unfortunately, is flawed, and there does not seem to be an easy way of fixing it.

vertex w , C is a simple cycle that passes through w , and Q is a simple path from w to v . Furthermore, $P \cap C = \{w\}$. If P contains k edges and C contains ℓ edges, then $d(u, v) = c(P) + \frac{\lambda^k}{1-\lambda^\ell} c(C)$. (Note that $c(Q)$ plays no role here.)

For every $u \in V$, let $d^\infty(u)$ be the infimum over the discounted costs of all infinite paths that start at u and *stay in the strongly connected component of u* . If there are no such paths, we let $d^\infty(u) = +\infty$. We can compute $d^\infty(u)$, for every $u \in V$, in $O(mn)$ time by first deleting edges that connect vertices in different strongly connected components, and then running the algorithm of the previous section for solving DMDPs.

We next describe a simple transformation that converts a general graph G into a graph G' of roughly the same size such that all discounted distances in G are realized by finite paths in G' . The graph G' is composed of two copies of G , with edges connecting vertices of the first copy to vertices of the second copy. More formally, if $G = (V, E, c)$, then $G' = (V', E', c')$, where $V' = \{v', v'' \mid v \in V\}$ and $E' = \{(u', v'), (u'', v'') \mid (u, v) \in E\} \cup \{(u', u'') \mid u \in V\}$. For every $(u, v) \in E$ we have $c'(u', v') = c(u, v)$ while $c'(u'', v'') = 0$. For every $u \in V$, we have $c'(u', u'') = d^\infty(u)$. For every $u, v \in V$, we clearly have $d(u, v) = d(u', v')$. If $d(u, v)$ is realized using a finite path in G , then it is also realized using a finite path in G' . For every $u, v \in V$, we also have $d(u', v'') \geq d(u, v)$. If $d(u, v)$ is obtained as the limit of a sequence of paths in G , then $d(u', v'') = d(u, v)$ and $d(u', v'')$ is attained in G' by a finite simple path.

5.1 The basic algorithm We apply the transformation described above and concentrate on finding discounted distances and shortest paths that are attained using simple paths. We next apply the following sampling lemma, which is similar to lemmas used by Ullman and Yannakakis [29] and Zwick [32], to a collection \mathcal{P} of at discounted shortest paths between all pairs of vertices in G . (For every $u, v \in V$, if $d(u, v)$ is attained using a (simple) finite path, we add one such discounted shortest path to \mathcal{P} .) The proof of the lemma is omitted.

LEMMA 5.1. *Let $G = (V, E)$ be a graph on n vertices and let \mathcal{P} be a collection of at most n^2 simple paths in G . Let $\pi = \langle v_1, v_2, \dots, v_n \rangle$ be a random permutation of the vertices of G . Then, with high probability, for every $P \in \mathcal{P}$ and every $1 \leq i < n$, if P is of length at least $h_i = \lceil \frac{4n \ln n}{i} \rceil$, then the index in π of at least one of the first h_i vertices in P is at most i .*

The basic algorithm works as follows. We choose a random permutation v_1, v_2, \dots, v_n of the vertices of G . The algorithm is then composed of n stages. In the i -th stage we compute $d(v_i, w)$, for every $w \in V$, assuming

that we already know $d(v_j, w)$, for every $1 \leq j < i$ and $w \in V$. (We actually compute $d(v_i, w)$ correctly only if it is attained by a finite path, but as explained above, this is enough.)

The i -th stage is organized as follows. We first use an adaptation of the classical Bellman-Ford algorithm to compute $d_k(v_i, w)$, for every $1 \leq k \leq h_i$ and $w \in V$,

$$d_k(v_i, w) = \min_{u: (u, w) \in E} d_{k-1}(v_i, u) + \lambda^{k-1} c(u, w).$$

This takes $O(h_i m) = O(\frac{mn \log n}{i})$ time. Next compute

$$d^{(1)}(v_i, w) = \min_{1 \leq k \leq h_i} d_k(v_i, w).$$

For a given vertex w , if $d(v_i, w)$ is attained using a path of at most h_i vertices, then $d^{(1)}(v_i, w) = d(v_i, w)$. If $d(v_i, w)$ is attained using a simple path that contains more than h_i edges, then by Lemma 5.1, with high probability, one of the first h_i vertices on this path, excluding v_i , has index at most i . As the path is simple, this index must be less than i . We thus compute

$$d^{(2)}(v_i, w) = \min_{1 \leq j < i} \min_{1 \leq k \leq h_i} d_k(v_i, v_j) + \lambda^{k-1} d(v_j, w).$$

This takes $O(i h_i n) = O(n^2 \log n)$ time. Finally, we set:

$$d(v_i, w) = \min\{d^{(1)}(v_i, w), d^{(2)}(v_i, w)\}.$$

The total running time of the i -th stage is $O(\frac{mn \log n}{i} + n^2 \log n)$, and the total running time is

$$O((\sum_{i=1}^n \frac{mn \log n}{i}) + n^3 \log n) = O(mn \log^2 n + n^3 \log n).$$

5.2 Improved algorithm We first observe that the computation of $d^{(2)}(v_i, w)$, for every $w \in V$, can be sped-up. In the basic algorithm, for every $1 \leq j < i$ and every $w \in V$, we consider all values of $1 \leq k \leq h_i$, explicitly compute $d_k(v_i, v_j) + \lambda^{k-1} d(v_j, w)$, and take the minimum over the h_i values computed. Consider now the set $\{d_k(v_i, v_j) + \lambda^{k-1} x \mid 1 \leq k \leq h_i\}$ of h_i linear functions. The function $\min_{1 \leq k \leq h_i} d_k(v_i, v_j) + \lambda^{k-1} x$ is a piecewise linear function having at most $h_i - 1$ breakpoints. As the slopes of the functions are given in sorted order, we can easily compute these breakpoints in $O(h_i)$ time. For every $w \in V$ we simply need locate $d(v_j, w)$ among these breakpoints. If we do that using binary search, the running time is reduced from $O(i h_i n)$ to $O(in \log n)$. We can reduce the running time even further to $O(in)$ by sorting $d(v_j, w)$, for all $w \in V$, and then merging this sorted list with the sorted list of the breakpoints. The total running is then reduced to $O(mn \log^2 n + n^3)$.

We can reduce the running time much further, if the graph is sparse. We proceed as follows. We run the algorithm as above for $i = 1, 2, \dots, \ell$, where

$\ell = \lceil (m \log n)^{1/2} \rceil$. The running time of the i -th stage is $O(\frac{mn \log n}{i} + in)$. The total running time of the first ℓ stages is therefore $O(mn \log^2 n + \ell^2 n) = O(mn \log^2 n)$. For $i = \ell + 1, \ell + 2, \dots, n$, we compute $d_k(v_i, w)$, for every $w \in V$ and for every $1 \leq k \leq h_\ell$. Note that we let k here range up to h_ℓ , and not up to h_i as before. In exchange, we let j in the computation of $d^{(2)}(v_i, w)$ range from 1 to ℓ , instead of from 1 to i . The running time of the i -th stage, for $\ell < i \leq n$, is now $O(\frac{mn \log n}{\ell} + \ell n) = O((m \log n)^{1/2} n)$ and the total running time of the improved algorithm is $O((m \log n)^{1/2} n^2 + mn \log^2 n)$.

6 Open problems

Is there an $o(mn^2)$ -time algorithm for discounted DMDPs in which each edge has its own discount factor? Is there an $\tilde{O}(mn)$ -time algorithm for the discounted all-pairs shortest paths problem?

References

- [1] D. Andersson and S. Vorobyov. Fast algorithms for monotonic discounted linear programs with two variables per inequality. TR NI06019-LAA, Isaac Newton Institute, Cambridge, United Kingdom, 2006.
- [2] R.E. Bellman. *Dynamic programming*. Princeton University Press, 1957.
- [3] D.P. Bertsekas. *Dynamic programming and optimal control*. Athena Scientific, second edition, 2001.
- [4] H. Björklund and S. Vorobyov. Combinatorial structure and randomized subexponential algorithms for infinite games. *Theoretical Computer Science*, 349(3):347–360, 2005.
- [5] E. Cohen and N. Megiddo. Improved algorithms for linear inequalities with two variables per inequality. *SIAM Journal on Computing*, 23(6):1313–1347, 1994.
- [6] A. Condon. The complexity of stochastic games. *Information and Computation*, 96:203–224, 1992.
- [7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, 2nd edition, 2001.
- [8] F. d'Epenoux. A probabilistic production and inventory problem. *Manag. Science*, 10(1):98–108, 1963.
- [9] C. Derman. *Finite state Markov decision processes*. Academic Press, 1972.
- [10] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8:109–113, 1979.
- [11] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.
- [12] V.A. Gurvich, A.V. Karzanov, and L.G. Khachiyan. Cyclic games and an algorithm to find minimax cycle means in directed graphs. *USSR Computational Mathematics and Mathematical Physics*, 28:85–91, 1988.
- [13] N. Halman. Simple stochastic games, parity games, mean payoff games and discounted payoff games are all LP-type problems. *Algorithmica*, 49(1):37–50, 2007.
- [14] D.S. Hochbaum and J. Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, 1994.
- [15] R.A. Howard. *Dynamic programming and Markov processes*. MIT Press, 1960.
- [16] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.
- [17] R.M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.
- [18] L.G. Khachiyan. A polynomial time algorithm in linear programming. *Soviet Math. Dokl.*, 20:191–194, 1979.
- [19] M.L. Littman. *Algorithms for sequential decision making*. PhD thesis, Brown University, 1996.
- [20] W. Ludwig. A subexponential randomized algorithm for the simple stochastic game problem. *Information and Computation*, 117(1):151–155, 1995.
- [21] O. Madani. *Complexity Results for Infinite-Horizon Markov Decision Processes*. PhD thesis, University of Washington, 2000.
- [22] O. Madani. On policy iteration as a Newton’s method and polynomial policy iteration algorithms. In *Proc. of the 18th AAAI*, pages 273–278, 2002.
- [23] O. Madani. Polynomial value iteration algorithms for deterministic MDPs. In *Proc. of the 18th UAI*, pages 311–318, 2002.
- [24] Y. Mansour and S.P. Singh. On the complexity of policy iteration. In *Proc. of the 15th UAI*, pages 401–408, 1999.
- [25] M. Melekopoglou and A. Condon. On the complexity of the policy improvement algorithm for markov decision processes. *ORSA Journal on Computing*, 6(2):188–192, 1994.
- [26] C.H. Papadimitriou and J.N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.
- [27] M.L. Puterman. *Markov decision processes*. Wiley, 1994.
- [28] L.S. Shapley. Stochastic games. *Proc. Nat. Acad. Sci. U.S.A.*, 39:1095–1100, 1953.
- [29] J.D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.
- [30] Y. Ye. A new complexity result on solving the Markov decision problem. *Mathematics of Operations Research*, 30(3):733–749, 2005.
- [31] N.E. Young, R.E. Tarjan, and J.B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21:205–221, 1991.
- [32] U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49:289–317, 2002.
- [33] U. Zwick and M.S. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1–2):343–359, 1996.