NEW METHODS FOR DYNAMIC PROGRAMMING OVER AN INFINITE TIME HORIZON

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF MANAGEMENT SCIENCE AND ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Michael Justin O'Sullivan
September 2002

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Arthur F. Veinott, Jr.
(Principal Co-Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Michael A. Saunders
(Principal Co-Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Benjamin Van Roy

Approved for the University Committee on Graduate Studies:

Abstract

Two unresolved issues regarding dynamic programming over an infinite time horizon are addressed within this dissertation. Previous research uses policy improvement to find a strong-present-value optimal policy in such systems, but the time complexity of policy improvement is not known. Here, a method is presented for substochastic systems that breaks the problem of finding a strong-present-value optimal policy into a number of smaller dynamic programming subproblems. Each of the subproblems may be solved using linear programming, giving the entire process a polynomial running time with regard to the size of the original dynamic programming problem. Also, for unique transition systems (a specialization of substochastic systems that includes general deterministic systems), other solution methods may be applied and the time complexity becomes strongly polynomial.

For normalized systems, policy improvement is still the method of choice. However, policy improvement requires the solution of linear systems that may not always have full rank. In a finite precision environment, the stability of solution methods for these linear systems is critical. One may simplify the computations associated with policy improvement by classifying the states and considering each class separately. Here, a method is presented that applies to any policy with substochastic classes. The method uses the state classification to break the linear system into many smaller linear systems that are either of full rank or rank-deficient by one. Each of the smaller linear systems is then solved in a numerically stable way.

During the development of the previous numerically stable method, the need for a sparse rank-revealing LU factorization became apparent. No such factorization exists in current literature. Here, a new sparse LU factorization is presented that uses a threshold form of complete pivoting. It is found to be rank-revealing for all but the most pathological matrices.

Acknowledgements

I would first like to thank my wife Síobhán and my two children Sophia and Croí. They continue to provide the love, support and inspiration necessary to my existence. I would also like to thank my parents, Alison and Mike, and my siblings, Susannah, John and Matthew. They have shaped me throughout my life and any success I have is a direct result of their love and support.

Professor Arthur F. Veinott, Jr. introduced me to the subject of dynamic programming and has guided me throughout my time at Stanford. I am always impressed by the extent of his knowledge. This dissertation is largely a direct result of his patience, insightful mind, and hard work. I would like to express my sincere appreciation for all his time and effort.

Professor Michael A. Saunders shakes my hand every time I see him. He has been a friend as well as an advisor during my time at Stanford. Anytime I had a question about linear algebra, he would not only supply the answer but also give useful insight into the problem. I would like to thank Mike for his kindness, thoughtfulness, and hard work throughout my time with him.

I would like to thank Professor Benjamin Van Roy for his input during the writing of this dissertation. His suggestions and comments were always useful and never failed to improve this work.

I would like to express my thanks to the faculty and staff of the Management Science and Engineering department at Stanford for their help and advice over the years. I would also like to thank Julie Ward and Hewlett-Packard for an exceptional work experience.

To my friends from Stanford and the Bay Area—you know who you are—I appreciate every moment we spent together. I hope I will see you again soon. Special thanks to those from the ex-colonies, New Zealand, South Africa and Australia, for helping me maintain my accent and sense of humor. Also, to my friends from the many football games I played, thanks for helping me retain my sanity through exercise.

Finally I would like to thank James Deaker and Karen McNay. James has been a good friend throughout my time at Stanford and I have never quite been able to thank him properly. Thanks mate. Karen adopted a poor, disorganized Kiwi boy and made me feel like part of the family. I hope Karen and her family realize how much this meant to me. Thanks to you all.

Contents

List of Figures

Chapter 1

Introduction

Many diverse applications exist for infinite-time-horizon dynamic programming (DP), e.g., developing medical treatment programs, determining harvest limits, and optimizing stock portfolios. This dissertation develops results in infinite-horizon DP and numerical linear algebra with applications to DP. The infinite-horizon results consist of polynomial-time and strongly polynomial-time methods for finding strong-present-value optimal policies in substochastic systems. The numerical linear algebra results include a stable method for calculating the present-value Laurent expansion coefficients for policies with substochastic classes. This method benefits from the other numerical linear algebra result, a sparse rank-revealing LU factorization.

The rest of this chapter is organized as follows. Section 1.1 discusses the appropriateness of strong-present-value optimality as the optimality criterion of choice. Previous research on substochastic systems is introduced in §1.2 and the lack of time complexity results discussed. Section 1.3 introduces the extension of the work on substochastic systems to normalized systems, and discusses implementation issues arising from a finite-precision environment. Finally, §1.4 summarizes the contributions of this dissertation in more detail.

1.1 Strong-Present-Value Optimality

Consider a finite-state-and-action Markov decision chain (referred to here as a DP system, or *system* for short). How does one solve the DP problem of selecting a policy to control this process? For finite time horizons, the total (expected) reward a policy earns over the lifetime of the process gives a measure with which to compare policies. However, an infinite time horizon may allow policies to earn infinite total reward. Previous research uses concepts such as maximum reward rate, present-value optimality, and strong-present-value optimality to differentiate between policies in the infinite horizon case.

The concept of maximum reward rate compares policies by looking at the average reward per period each policy earns over the lifetime of the process. However, this optimality concept ignores the initial (possibly transient) behavior of a policy, focusing on long-term rewards. On the other hand, present-value optimality assumes a (positive) interest rate and compares policies by discounting all future rewards to the present. This optimality concept focuses on short-term rewards generated by a policy, since the importance of any long-term behavior will be reduced by discounting. Under strong-present-value optimality, optimal policies are those that simultaneously maximize present-value for all small (positive) interest rates. Such policies also earn maximum reward rate. Strong-present-value optimality considers short, intermediate, and long-term behavior when comparing policies.

1.2 Substochastic Systems

In a *substochastic system*, the transition matrix for each stationary policy is substochastic. Blackwell [Bla62] introduces the problem of finding a strong-present-value optimal policy, i.e., a policy that has maximum present value for all sufficiently small positive interest rates, and shows non-constructively that a stationary strong-present-value optimal policy exists. Miller and Veinott [MV69] give a constructive proof by developing a policy-improvement method for finding such a policy. This method exploits the fact that the present value of the expected rewards that a policy earns has a Laurent expansion in the interest rate.

Previous work builds towards strong-present-value optimality by developing the weaker concept of finding a stationary policy that is n-optimal, i.e., lexicographically maximizes the first $n+2$ coefficients of the Laurent expansion of the present value. Where $n = -1$, this is the classic concept of finding a stationary maximum-reward-rate policy. Howard [How60][1] and Blackwell [Bla62] show how to solve that problem by policy improvement. Manne [Man60], de Ghellinck [dG60], Derman [Der62] and Denardo [Den70b] develop one linear programming method for doing this, and Balinski [Bal61], Denardo and Fox [DF68], and Hordijk and Kallenberg [HK79] give another. Blackwell [Bla62] introduces the stronger concept of finding a stationary 0-optimal policy. Veinott [Vei66] shows how to solve this problem by policy improvement and Denardo [Den70a] shows how to do this by linear programming.

Veinott [Vei66, Vei68, Vei69, Vei74] introduces the problem of finding a stationary n-optimal policy and shows its equivalence to finding a stationary n-present-value (resp., n-Cesàro-overtaking) optimal policy. A stationary policy is strong-present-value optimal if and only if it is S-optimal where S is the number of states. The papers [Vei69, Vei74] refine the policy-improvement algorithm of [MV69] to find a stationary n-optimal policy and suggest that a good way to implement that algorithm is to find a stationary m-optimal policy for $m = -1, \ldots, n$ in that order.

Veinott [Vei69] and Avrachenkov and Altman [AA98] show how to find an n-optimal policy in specialized systems (transient and irreducible, respectively) using a sequence of smaller problems

[1] See [Vei66, pp. 1291–1293] for a development that fills a lacunae in the proof in [How60].

that may be solved using linear programming.

Denardo [Den71, p. 491] sketches a method for finding an n-optimal policy using either policy improvement or a combination of linear programming and some intermediate algorithms to solve a sequence of simpler DP problems. However, his method is subject to counterexample. Indeed, appendix A gives an example in which his method fails to find a 1-optimal policy.

No previous work successfully shows how to use linear programming alone to find either an n-optimal policy for $n > 0$ or a strong-present-value optimal policy. Since the time complexity of policy improvement is unknown, the time required to find a strong-present-value optimal policy is not currently known.

1.3 Normalized Systems

In a *normalized system*, the transition matrix of every stationary policy has spectral radius not exceeding one. As Rothblum [Rot75] notes, Blackwell's [Bla62] proof of existence of a stationary strong-present-value optimal policy for substochastic systems extends in a straightforward way to normalized systems. Rothblum [Rot75] extends the methods from Miller and Veinott [MV69] and Veinott [Vei69, Vei74] by giving an augmented Laurent expansion of the present value and generalizing their policy improvement method to normalized systems.

The (more general) policy improvement requires the coefficients of the (augmented) Laurent expansion. These coefficients are the unique solution of a set of linear equations (identical to those from Veinott [Vei69] except for the number of arbitrary variables). Veinott [Vei69] shows how to efficiently solve these linear equations (for substochastic systems) by identifying recurrent classes, solving within each such (stochastic, irreducible) class by repeated application of a Gaussian elimination matrix, and using these solutions to solve the remainder of the system. Rothblum [Rot75] notes that Veinott's [Vei69] method may be extended to normalized systems, but this requires extra work to partition the system into communicating classes and identify which classes are recurrent. No previous work discusses the numerical properties of the linear equations (although Veinott [Vei69] uses the linear dependence of these equations within recurrent classes) or the numerical stability of solution methods.

1.4 Dissertation Outline

Chapter 2 presents a new method for finding an n-optimal (resp., strong-present-value optimal) policy within a substochastic system by solving a sequence of $3n+5$ simpler, well-studied DP subproblems. Previous work shows that these subproblems may be solved using either policy improvement or linear programming. Since linear programming can be solved in time that is polynomial in the size of the problem, this method finds an n-optimal (resp., strong-present-value optimal) policy in polynomial time when using linear programming.

For a special refinement of substochastic systems, referred to here as *unique transition systems*, this method finds an n-optimal (resp., strong-present-value optimal) policy in strongly polynomial time. This result is especially appealing because the refinement includes general deterministic systems.

Chapter 3 presents a new method similar to Veinott's [Vei69] for determining the coefficients of the Laurent expansion in a system with substochastic classes. This method reduces the problem to that of finding the coefficients within a number of irreducible, substochastic systems. By partitioning the state space into communicating classes and following the induced dependence partial ordering, the coefficients for the entire system are found by considering each class separately. Within each (irreducible, substochastic) class, the new method determines if the class is recurrent or transient using a rank-revealing LU (RRLU) factorization, then computes the coefficients by solving the linear equations (for that class) in a numerically stable way.

The research contained in Chapter 4 arose during the development of the numerically stable computations from Chapter 3. In large, sparse DP problems, solving the linear equations (for each class) may require factorizing a large, sparse matrix with a possible singularity. Since many of these coefficient matrices are encountered within each iteration, a fast, sparse RRLU factorization is desirable. Chan [Cha84], Hwang, Lin and Yang [HLY92], and Hwang and Lin [HL97] give dense RRLU factorizations. The LUSOL package of Gill, Murray, Saunders, and Wright [GMSW87] contains a fast, sparse LU factorization using threshold partial pivoting. This is sometimes, but not always, rank-revealing. Chapter 4 shows that adding a threshold form of complete pivoting to LUSOL gives a sparse RRLU factorization. This is compared with several other factorizations using different measures.

Chapter 2

Polynomial-Time Dynamic Programming

2.1 Introduction

This chapter presents a method that finds a strong-present-value optimal policy for a finite-state-and-action infinite-horizon Markov decision chain in polynomial time. Using the weaker notion of n-optimality, this new method starts with an arbitrary policy and sequentially improves the policy until a strong-present-value optimal policy is found. The problem of finding an n-optimal policy from an $(n-1)$-optimal policy is decomposed into three simpler subproblems, each of which may be solved in polynomial time using linear programming. If n is the number of states the method yields a strong-present-value optimal policy in polynomial time.

Section 2.2 formulates the problem of finding a strong-present-value optimal policy. That section defines the system, introduces relevant optimality concepts and conditions, and discusses previous approaches for finding such policies. It also introduces the subproblems at the core of the new method. The method itself is introduced in §2.3, taking the form of a theorem that demonstrates how to move from an $(n-1)$-optimal policy to a n-optimal policy using three simple (dynamic programming) subproblems. The extension of this theorem to find a strong-present-value optimal policy in polynomial time is then given. Next, by classifying the states, §2.4 shows how to efficiently implement the new method. The running time of this method becomes strongly polynomial for unique transition systems, a refinement of the original system definition that includes standard deterministic finite-state-and-action infinite-horizon dynamic programs. Section 2.5 describes three different forms of these systems, and gives the new running time in each case. Finally, §2.6 compares earlier decomposition methods with the new method and §2.7 summarizes the contributions within this chapter.

2.2 Preliminaries

2.2.1 Formulation

Consider a *substochastic system* observed in *periods* $1, 2, \ldots$. In each period, the system is in one of a finite collection $\mathcal{S}$ of S *states* or is in the *stopped state*. In each state $s \in \mathcal{S}$, a finite set $\mathcal{A}_s$ of *actions* is available. The set of state-action pairs $\{(s,a) \mid s \in \mathcal{S}, a \in \mathcal{A}_s\}$ has cardinality A. If one observes in a period that the system is in state $s \in \mathcal{S}$ and chooses action $a \in \mathcal{A}_s$, the system earns expected *reward* $r(s,a)$ and next period moves to state $t \in \mathcal{S}$ with probability $p(t|s,a) \geq 0$ and the stopped state with probability $1 - \sum_{t \in \mathcal{S}} p(t|s,a)$. Once a system enters the stopped state it remains there and earns no further reward.

Since Blackwell [Bla62] proves the existence of a stationary strong-present-value optimal policy, it suffices to restrict attention to stationary policies within this chapter. Throughout, a *(stationary) policy* is a function δ that assigns an action $\delta_s \in \mathcal{A}_s$ to each state $s \in \mathcal{S}$. Each policy δ induces an S-vector $r_\delta \equiv (r(s,\delta_s))$ of expected rewards and an $S \times S$ *transition matrix* $P_\delta \equiv (p(t|s,\delta_s))$. Let Δ be the set of all policies.

The N-period transition matrix for a policy δ is the N^{th} power P_δ^N of P_δ. This leads to the definition of a *stationary transition matrix* $P_\delta^* \equiv \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{\infty} P_\delta^{i-1}$ for a policy δ. Now each policy δ partitions $\mathcal{S}$ into the *transient* states $\mathcal{T}_\delta \equiv \{t \in \mathcal{S} \mid P_{\delta st}^* = 0 \text{ for all } s \in \mathcal{S}\}$ under δ and the *recurrent* states $\mathcal{R}_\delta \equiv \mathcal{S} \setminus \mathcal{T}_\delta$ under δ.[1] If $\mathcal{T}_\delta = \mathcal{S}$ (i.e., $P_\delta^* = 0$), then δ is said to be *transient*.

2.2.2 Optimality Concepts and Conditions

Strong-present-value optimality. Suppose that rewards carried from one period to the next earn interest at the rate $100\rho\%$ ($\rho > 0$) and let $\beta \equiv \frac{1}{1+\rho}$ be the *discount factor*. The *present value* V_δ^ρ of a policy δ is the (expected) present value of the rewards that δ earns in each period discounted to the beginning of period 0, i.e., $V_\delta^\rho \equiv \sum_{N=1}^{\infty} \beta^N P_\delta^{N-1} r_\delta$. A policy δ is *present-value optimal* if $V_\delta^\rho \geq V_\gamma^\rho$ for all $\gamma \in \Delta$. Finally, δ is *strong-present-value optimal* if it is present-value optimal for all sufficiently small ρ.

***n*-Optimality.** It is computationally challenging to discern directly whether or not a policy is strong-present-value optimal. However, building a sequence of n-optimal policies is more efficient and attains strong-present-value optimality (when $n = S$).

A policy δ is *n-present-value optimal* if

$$\lim_{\rho \downarrow 0} \rho^{-n} \left(V_\delta^\rho - V_\gamma^\rho \right) \geq 0 \text{ for all } \gamma \in \Delta. \tag{2.1}$$

[1] The classification of states as transient or recurrent under a given policy leads to the concept of recurrent classes defined in §2.4.

Evidently

$$V_\delta^\rho = \beta r_\delta + \beta P_\delta V_\delta^\rho. \tag{2.2}$$

Also, V_δ^ρ has the Laurent expansion

$$V_\delta^\rho = \sum_{n=-1}^{\infty} \rho^n v_\delta^n \tag{2.3}$$

in small $\rho > 0$ [MV69]. By substituting (2.3) into (2.2), multiplying by $1+\rho$ and equating coefficients of like powers of ρ, ones sees that $V^{n+1} \equiv \left(v^{-1}, v^0, \ldots, v^{n+1}\right) = \left(v_\delta^{-1}, v_\delta^0, \ldots, v_\delta^{n+1}\right)$ satisfies

$$r_\delta^j + Q_\delta v^j = v^{j-1}, j = -1, 0, \ldots, n+1, \tag{2.4}$$

where $Q_\delta = P_\delta - I$, $r_\delta^0 = r_\delta$, $r_\delta^j = 0, j \neq 0$, and $v_\delta^{-2} = 0$ [MV69]. Conversely, if the matrix $V^{n+1} \equiv \left(V^n, v^{n+1}\right)$ satisfies (2.4) then $V^n = V_\delta^n \equiv \left(v_\delta^{-1}, v_\delta^0, \ldots, v_\delta^n\right)$ [Vei69]. Thus, (2.4) uniquely determines the vector $V^n = V_\delta^n$, but not v^{n+1}.

Writing $B \succeq C$ for two matrices of like size means that each row of $B - C$ is lexicographically nonnegative. Combining (2.1) and (2.3), a policy δ is n-present-value optimal if and only if δ is n-optimal, i.e., $V_\delta^n \succeq V_\gamma^n$ for all $\gamma \in \Delta$.

Denote the set of n-optimal policies by Δ_n and notice that the n-optimal sets are nested, i.e., $\Delta \supseteq \Delta_{-1} \supseteq \Delta_0 \supseteq \Delta_1 \supseteq \cdots$. Extending [MV69], [Vei69] shows that there exists an $1 \leq m \leq S$ such that $\Delta \supseteq \Delta_{-1} \supseteq \cdots \supseteq \Delta_m = \Delta_{m+1} = \cdots$. Moreover, an S-optimal policy is strong-present-value optimal.

2.2.3 Policy Improvement

Veinott [Vei69, Vei74] refines the policy improvement method given in [MV69] to find an n-optimal policy as follows. For $\gamma, \delta \in \Delta$, let

$$g_{\gamma\delta}^j \equiv r_\gamma^j + Q_\gamma v_\delta^j - v_\delta^{j-1}, \quad j = -1, 0, \ldots \tag{2.5}$$

and $G_{\gamma\delta}^n \equiv (g_{\gamma\delta}^{-1}, \ldots, g_{\gamma\delta}^n)$ for $n \geq -1$. If δ is n-optimal, then

$$G_{\gamma\delta}^n \preceq 0 \text{ for all } \gamma \in \Delta. \tag{2.6}$$

Conversely, if

$$G_{\gamma\delta}^{n+1} \preceq 0 \text{ for all } \gamma \in \Delta, \tag{2.7}$$

then δ is n-optimal. Therefore, (2.6) is necessary and (2.7) is sufficient for n-optimality of δ. We say that δ satisfies the n-optimality conditions if (2.7) holds. If $G_{\gamma\delta}^n \succ 0$ for some $\gamma \in \Delta$ and $\gamma_s = \delta_s$ wherever $G_{\gamma\delta s}^n = 0$, call γ an n-improvement of δ. In that event $V_\gamma^n \succ V_\delta^n$. If δ has no $(n+1)$-improvement, then it satisfies the n-optimality conditions.

Let $\mathcal{E}_\delta^{-2} \equiv \Delta$ and $\mathcal{E}_\delta^n \equiv \{\gamma \in \Delta : G_{\gamma\delta}^n = 0\}$ for $n \geq -1$, so $\mathcal{E}_\delta^n = \{\gamma \in \mathcal{E}_\delta^{n-1} : g_{\gamma\delta}^n = 0\}$ for $n \geq -1$. Since $V^n = V_\delta^n$ satisfies (2.4) for $-1 \leq j \leq n$, $\delta \in \mathcal{E}_\delta^n$. Also, for any two policies γ and δ, Lemma 9 of [Vei69, p. 1648] implies that $V_\gamma^n - V_\delta^n$ is a linear function of $G_{\gamma\delta}^{n+1}$. In particular,

$$\gamma \in \mathcal{E}_\delta^n \text{ implies that } V_\gamma^{n-1} = V_\delta^{n-1} \text{ and } v_\gamma^n - v_\delta^n = P_\gamma^* g_{\gamma\delta}^{n+1} = P_\gamma^* \left(r_\gamma^{n+1} - v_\delta^n \right) \tag{2.8a}$$

and so

$$V_\gamma^{n-1} = V_\delta^{n-1} \text{ implies that } \gamma \in \mathcal{E}_\delta^{n-1} \text{ and } 0 = v_\gamma^{n-1} - v_\delta^{n-1} = P_\gamma^* g_{\gamma\delta}^n. \tag{2.8b}$$

It is immediate from the first assertion in (2.8a) and the definitions involved that

$$\text{if } \delta \text{ is } (n-1)\text{-optimal, then } \mathcal{E}_\delta^n \subseteq \Delta_{n-1} \subseteq \mathcal{E}_\delta^{n-1}. \tag{2.9}$$

Policy improvement may be implemented inductively as follows [Vei69, Vei74]. Given a policy δ that is $(n-1)$-optimal, search $\mathcal{E}_\delta^{n-1}$ for an $(n+1)$-improvement γ of δ (so γ is $(n-1)$-optimal), replace δ by γ, and repeat this procedure until a policy is found with no $(n+1)$-improvement. The last policy satisfies the n-optimality conditions and so is n-optimal.

2.2.4 Maximum-Transient-Value Policies

If δ is transient (see §2.2.1), its value $V_\delta = \sum_{i=1}^\infty P_\delta^{i-1} r_\delta$ is finite, and δ has *maximum* transient value if $V_\delta \geq V_\gamma$ for all transient $\gamma \in \Delta$. The maximum-transient-value problem is to find a transient policy with maximum value among all such policies. Note that there is no requirement that every policy be transient. The paper [EV75] establishes the following facts about this problem. A policy δ has maximum transient value if and only if δ is transient and V_δ is the least fixed point of the optimal return operator $\mathcal{R}$ defined by $\mathcal{R}V \equiv \max_{\gamma \in \Delta} (r_\gamma + P_\gamma V)$ for $V \in \mathbb{R}^S$. Such a policy exists if and only if there is a transient policy and $\mathcal{R}$ has an excessive point, i.e., a $V \in \mathbb{R}^S$ for which $V \geq \mathcal{R}V$. It is possible to find a maximum-transient-value policy either by policy iteration or by solving a linear program with S rows and A columns similar to that of d'Epenoux [d'E60].

2.2.5 Maximum-Reward-Rate Policies

The reward rate, i.e., the expected reward per period, of δ is $v_\delta^{-1} = P_\delta^* r_\delta$. The maximum-reward-rate problem is to find a policy with maximum reward rate. This problem is well-studied and previous work shows that there is such a policy and it can be found by policy iteration [How60, Bla62] or by solving a linear program [Bal61, DF68, HK79] with $2S$ rows and $2A$ columns.

2.3 Strong-Present-Value Optimality in Polynomial Time

Given an $(n-1)$-optimal policy, this section shows how to construct an n-optimal policy. It follows that the construction of such a policy takes polynomial time, so a strong-present-value optimal policy may be found in polynomial time.

To that end, let $(v_*^{-1}, v_*^0, v_*^1, \dots)$ be the lexicographic maximum of $(v_\delta^{-1}, v_\delta^0, v_\delta^1, \dots)$ over $\delta \in \Delta$, $V_*^n \equiv (v_*^{-1}, \dots, v_*^n)$ and $\mathcal{E}_*^n \equiv \{\gamma \in \mathcal{E}_*^{n-1} : r_\gamma^n + Q_\gamma v_*^n - v_*^{n-1} = 0\}$ for $n \geq -1$ where $v_*^{-2} \equiv 0$ and $\mathcal{E}_*^{-2} \equiv \Delta$.

For each policy ζ, observe that $\mathcal{E}_\zeta^n$ has the product property, i.e., is a product $\mathcal{E}_\zeta^n = \bigtimes_{s \in \mathcal{S}} \mathcal{E}_{\zeta s}^n$ of subsets $\mathcal{E}_{\zeta s}^n \subseteq \mathcal{A}_s$ of actions for each state $s \in \mathcal{S}$. Let $\underline{\mathcal{E}}_\zeta^n$ be a set of policies such that

(i) $\underline{\mathcal{E}}_\zeta^n$ has the product property $\underline{\mathcal{E}}_\zeta^n = \bigtimes_{s \in \mathcal{S}} \underline{\mathcal{E}}_{\zeta s}^n$ for some subsets $\emptyset \neq \underline{\mathcal{E}}_{\zeta s}^n \subseteq \mathcal{E}_{\zeta s}^n$ for all $s \in \mathcal{S}$, and

(ii) for each $s \in \mathcal{S}$, $\underline{\mathcal{E}}_{\zeta s}^n$ contains each action that is used by some policy in $\mathcal{E}_\zeta^n$ for which state s is recurrent under that policy.

Thus, $\emptyset \neq \underline{\mathcal{E}}_\zeta^n \subseteq \mathcal{E}_\zeta^n$. The simplest example of such a set $\underline{\mathcal{E}}_\zeta^n$ is $\mathcal{E}_\zeta^n$ itself. Section 2.4 gives other examples that lead to substantially more efficient algorithms for solving the maximum-reward-rate subproblem in **(b)** of Theorem 2.1 below. Note that $\zeta \notin \underline{\mathcal{E}}_\zeta^n$ is possible.

Theorem 2.1 Sequential Decomposition into Subproblems.
Suppose $n \geq -1$.

(a) $(n-1)$-Optimality Conditions. *If δ is $(n-1)$-optimal, then $v^n = v_\zeta^n$ is the least solution of*

$$v^n = \max_{\gamma \in \mathcal{E}_*^{n-1}} \left(r_\gamma^n - v_*^{n-1} + P_\gamma v^n \right) \vee v_\delta^n \tag{2.10}$$

for some $\zeta \in \mathcal{E}_*^{n-1}$ that is transient on the states s for which $v_{\zeta s}^n > v_{\delta s}^n$ and that agrees with δ otherwise. Moreover, each such $\gamma = \zeta$ maximizes v_γ^n over all such γ and satisfies the $(n-1)$-optimality conditions.[2]

(b) n-Optimality for Recurrent States. *If ζ satisfies the $(n-1)$-optimality conditions, then*

$$\max_{\gamma \in \underline{\mathcal{E}}_\zeta^n} v_\gamma^n = v_\zeta^n + \max_{\gamma \in \underline{\mathcal{E}}_\zeta^n} P_\gamma^* \left(r_\gamma^{n+1} - v_\zeta^n \right). \tag{2.11}$$

Moreover, the sets of maximizers on both sides of (2.11) *are nonempty and coincide. Also, each such common maximizer $\gamma = \eta$ is $(n-1)$-optimal and is n-optimal on each state that is recurrent under some n-optimal policy.*

[2]It is not efficient to implement part **(a)** when $n = -1$. In that case, it is better to omit **(a)** and begin with a modification of part **(b)** that omits the hypothesis about ζ, replaces v_ζ^{-1} by 0 and $\mathcal{E}_\zeta^{-1}$ by Δ. Then the modified version of **(b)** evidently holds. If also one sets $\underline{\mathcal{E}}_\zeta^{-1} = \Delta$, then the policy $\gamma = \eta$ found in **(b)** is -1-optimal, so part **(c)** is unnecessary. However, as §2.4 discusses, it is necessary to solve subproblem **(c)** for other (often better) choices of $\underline{\mathcal{E}}_\zeta^{-1}$.

(c) n-Optimality. *If η is $(n-1)$-optimal and is n-optimal on each state that is recurrent under some n-optimal policy, then $v^n = v^n_\theta$ is the least solution of*

$$v^n = \max_{\gamma \in \mathcal{E}_*^{n-1}} \left(r^n_\gamma - v^{n-1}_* + P_\gamma v^n \right) \vee v^n_\eta \tag{2.12}$$

for some $\theta \in \mathcal{E}_*^{n-1}$ that is transient on the set of states s for which $v^n_{\theta s} > v^n_{\eta s}$ and that agrees with η otherwise. Moreover, each such θ is n-optimal.

Role of the Three Subproblems It seems useful to explain briefly the role of each of the three subproblems therein. Consider a policy δ that is $(n-1)$-optimal. Then it is necessary to find a γ in Δ_{n-1} that maximizes v^n_γ over Δ_{n-1}. Unfortunately, this is generally hard to do for two reasons. First, it is difficult to find Δ_{n-1}. Second, even if one has Δ_{n-1}, it may not have the product property, which significantly complicates the problem of optimizing over that set. However, it follows from (2.9) that Δ_{n-1} has inner and outer approximations $\mathcal{E}^n_\delta$ and $\mathcal{E}^{n-1}_\delta = \mathcal{E}^{n-1}_*$, respectively, that are easy to compute, have the product property and can be used as follows.

First solve the maximum-transient-value subproblem in **(a)** to find a policy $\gamma = \zeta$ that maximizes v^n_γ over the γ in $\mathcal{E}^{n-1}_*$ that are transient on states where γ differs from δ. The policy ζ also satisfies the $(n-1)$-optimality conditions. Now solve the maximum-reward-rate subproblem in **(b)** to find a policy $\gamma = \eta$ that maximizes v^n_γ over $\underline{\mathcal{E}}^{\,n}_\zeta$. Since ζ satisfies the $(n-1)$-optimality conditions ($(n-1)$-optimality is not enough), it turns out that η is n-optimal on the states where some n-optimal policy is recurrent (though it doesn't find those states). Finally, solve the maximum-transient-value subproblem in **(c)** to find a policy $\gamma = \theta$ that maximizes v^n_γ over γ in $\mathcal{E}^{n-1}_*$ that are transient on states where γ differs from η. The policy θ is n-optimal.

The following corollary is a direct result of the subproblem decomposition of Theorem 2.1.

Corollary 2.2 Polynomial Running Time. *The problem of finding an n-optimal, and hence a strong-present-value optimal, policy is solvable in polynomial time.*

Proof. One may find an n-optimal policy by sequentially solving subproblems **(a)**, **(b)** (with $\underline{\mathcal{E}}^{\,n}_\zeta = \mathcal{E}^n_\zeta$), and **(c)** of Theorem 2.1 to find an m-optimal policy in the order $m = -1, 0, \ldots, n$. Each of these $3n+5$ subproblems is solvable using a linear program whose size is polynomial in the size of the dynamic program. Khachian [Kha79] shows that a linear program is solvable in time that is polynomial in its size. Also, the size of the solution of a linear program is polynomial in the size of the linear program itself. The size of each linear program for solving subproblems **(a)**, **(b)**, and **(c)** depends only on the size of the dynamic program and the solutions of the previous linear program. Therefore, the size of each linear program for solving subproblems **(a)**, **(b)**, and **(c)** is polynomial in the size of the dynamic program, so the time required to solve each subproblem is polynomial in the size of the dynamic program. Therefore, the time to find an n-optimal policy is polynomial in the size of the dynamic program. Also, since an S-optimal policy is strong-present-value optimal, a strong-present-value optimal policy may be found in polynomial time. ■

The rest of this section contains a proof for Theorem 2.1 and an example that demonstrates that the three subproblems produce distinct solutions.

Proof of Theorem 2.1. Suppose that α is n-optimal. Also, for parts **(b)** and **(c)**, let $R \equiv \mathcal{R}_\alpha$ (resp., $T \equiv \mathcal{T}_\alpha$) be the set of states where α is recurrent (resp., transient). Remember the sets R and T partition $\mathcal{S}$.

(a) Suppose δ is $(n-1)$-optimal. Define a new system by first restricting the policies to $\mathcal{E}_*^{n-1}$, letting the one-period rewards be $r_\gamma^n - v_*^{n-1}$ for $\gamma \in \mathcal{E}_*^{n-1}$, and appending a new action in every state $s \in \mathcal{S}$ that stops and earns the one-period reward $v_{\delta s}^n$. The policy that stops in each state and period is transient in the new system. Then for all $\gamma \in \Delta$, $V_\alpha^n \succeq V_\gamma^n$ and, from (2.6), $G_{\gamma\alpha}^n \preceq 0$. Also $V_\delta^{n-1} = V_\alpha^{n-1}$, so for $\gamma \in \mathcal{E}_*^{n-1} = \mathcal{E}_\alpha^{n-1}$, $G_{\gamma\alpha}^{n-1} = 0$ and $g_{\gamma\alpha}^n \le 0$. The last fact and $v_\alpha^n \ge v_\delta^n$ imply that v_α^n is an excessive point of the operator on the right-hand side of (2.10). Therefore, from [EV75], there is a transient policy κ in the new system whose value V_κ therein is the least solution of (2.10) and majorizes the value of every other transient policy in that system.

Next construct a policy $\zeta \in \mathcal{E}_*^{n-1}$ in the original system with ζ being $(n-1)$-optimal and $v_\zeta^n = V_\kappa$. To that end, let $E = \{s \in \mathcal{S} : V_{\kappa s} = v_{\delta s}^n\}$, so $V_{\kappa G} \gg v_{\delta G}^n$ where $G \equiv \mathcal{S} \setminus E$. Define ζ by $\zeta_E \equiv \delta_E$ and $\zeta_G \equiv \kappa_G$. Now $P_{\delta EG} = 0$, for otherwise $P_{\delta EG} > 0$ and so $V_{\kappa E} = v_{\delta E}^n = r_{\delta E}^n - v_{*E}^{n-1} + P_{\delta EE} v_{\delta E}^n + P_{\delta EG} v_{\delta G}^n < r_{\delta E}^n - v_{*E}^{n-1} + P_{\delta EE} V_{\kappa E} + P_{\delta EG} V_{\kappa G} \le V_{\kappa E}$, which is a contradiction. Therefore, by suitably permuting the states,

$$P_\zeta = \begin{pmatrix} P_{\kappa GG} & P_{\kappa GE} \\ 0 & P_{\delta EE} \end{pmatrix} \text{ and } P_\zeta^* = \begin{pmatrix} P_{\kappa GG}^* & P_{\zeta GE}^* \\ 0 & P_{\delta EE}^* \end{pmatrix} = \begin{pmatrix} 0 & P_{\zeta GE}^* \\ 0 & P_{\delta EE}^* \end{pmatrix}, \tag{2.13}$$

the last since κ is transient, so ζ is transient on G. Also from (2.10), the definition of κ and $\delta \in \mathcal{E}_\delta^{n-1} = \mathcal{E}_*^{n-1}$, it follows that $\zeta \in \mathcal{E}_*^{n-1}$. These facts and (2.8a) imply that $V_\zeta^{n-2} = V_*^{n-2}$ and $v_\zeta^{n-1} - v_*^{n-1} = P_\zeta^* g_{\zeta\delta}^n = P_{\zeta\mathcal{S}E}^* g_{\delta\delta E}^n = 0$, so ζ is $(n-1)$-optimal. Now since $P_{\zeta GG} = P_{\kappa GG}$ is transient, $V = V_\kappa$ is the unique solution of

$$V = \begin{pmatrix} r_{\zeta G}^n - v_{*G}^{n-1} \\ v_{\delta E}^n \end{pmatrix} + \begin{pmatrix} P_{\zeta GG} & P_{\zeta GE} \\ 0 & 0 \end{pmatrix} V. \tag{2.14}$$

However, $V = v_\zeta^n$ also satisfies this equation. To see this for states in G, substitute $v_{\zeta G}^{n-1} = v_{*G}^{n-1}$ into $g_{\zeta\zeta G}^n = 0$, while for states in E, recall that $P_{\zeta EG} = P_{\delta EG} = 0$ and so $v_{\zeta E}^n = v_{\delta E}^n$. Thus since the above equation has a unique solution, $V_\kappa = v_\zeta^n$.

Since $\zeta \in \mathcal{E}_*^{n-1} = \mathcal{E}_\delta^{n-1}$, $V_\zeta^{n-1} = V_\delta^{n-1}$ by (2.8a), so $G_{\gamma\zeta}^{n-1} = G_{\gamma\delta}^{n-1} \preceq 0$ for all $\gamma \in \Delta$ by (2.6). Also, since v_ζ^n satisfies (2.10), $g_{\gamma\zeta}^n \le 0$ for all $\gamma \in \mathcal{E}_*^{n-1} = \mathcal{E}_\zeta^{n-1}$, i.e., for all γ with $G_{\gamma\zeta}^{n-1} = 0$. Hence $G_{\gamma\zeta}^n \preceq 0$ for all $\gamma \in \Delta$, i.e., ζ satisfies the $(n-1)$-optimality conditions.

(b) Suppose ζ satisfies the $(n-1)$-optimality conditions. Now since $\emptyset \neq \underline{\mathcal{E}}_\zeta^n \subseteq \mathcal{E}_\zeta^n$ and $\underline{\mathcal{E}}_\zeta^n$ has the product property, it follows from (2.8a) that (2.11) holds and the sets of maximizers on both sides thereof coincide. Thus, from [How60] and [Bla62], there is a $\gamma = \eta$ maximizing the right-hand, and so the left-hand, side of (2.11) because the former is a maximum-reward-rate problem with one-period rewards $r_\gamma^{n+1} - v_\zeta^n$ for $\gamma \in \underline{\mathcal{E}}_\zeta^n$. Since $\eta \in \underline{\mathcal{E}}_\zeta^n \subseteq \mathcal{E}_\zeta^n$, $V_\eta^{n-1} = V_\zeta^{n-1}$ by (2.8a), so η is $(n-1)$-optimal. Now $G_{\alpha\zeta}^{n-1} = 0$, from $V_\alpha^{n-1} = V_\zeta^{n-1}$ (both $(n-1)$-optimal) and (2.8b), so $g_{\alpha\zeta}^n \leq 0$ because ζ satisfies the $(n-1)$-optimality conditions. Also, (2.8b) gives $0 = v_\alpha^{n-1} - v_\zeta^{n-1} = P_\alpha^* g_{\alpha\zeta}^n = P_{\alpha\mathcal{S}R}^* g_{\alpha\zeta R}^n$, so $g_{\alpha\zeta R}^n = 0$ (since $P_\alpha^* \geq 0$). Hence $G_{\alpha\zeta R}^n = 0$ and there is a $\lambda \in \underline{\mathcal{E}}_\zeta^n$ with $\lambda_R = \alpha_R$ and λ recurrent on R because α is recurrent thereon. Therefore $v_\lambda^n \leq v_\eta^n$ by (2.11). Also $P_{\alpha RT} = 0$, so $v_{\alpha R}^n = v_{\lambda R}^n \leq v_{\eta R}^n \leq v_{\alpha R}^n$, whence $v_{\eta R}^n = v_{\alpha R}^n$.

(c) Suppose η is $(n-1)$-optimal and is n-optimal on each state that is recurrent under some n-optimal policy. Then from **(a)** of the Theorem, there is a $\theta \in \mathcal{E}_*^{n-1}$ satisfying the assertions of **(c)** of the Theorem except perhaps for $\theta \in \Delta_n$. To show the last fact, observe that since v_θ^n satisfies (2.12), $v_\alpha^n \geq v_\theta^n \geq v_\eta^n$. Moreover, by hypothesis $v_{\eta R}^n = v_{\alpha R}^n$ and so $v_{\theta R}^n = v_{\alpha R}^n$. Also since v_θ^n satisfies (2.12) and $\alpha \in \Delta_n \subseteq \mathcal{E}_*^{n-1}$, $v_\theta^n \geq r_\alpha^n - v_*^{n-1} + P_\alpha v_\theta^n = v_\alpha^n + P_\alpha(v_\theta^n - v_\alpha^n)$ implying $v_\theta^n - v_\alpha^n \geq P_\alpha(v_\theta^n - v_\alpha^n)$. Iterating this inequality yields $v_\theta^n - v_\alpha^n \geq P_\alpha^N(v_\theta^n - v_\alpha^n)$ for $N = 0, 1, \ldots$. Taking the $(\mathrm{C}, 1)$ limit of the last inequality yields $v_\theta^n - v_\alpha^n \geq P_\alpha^*(v_\theta^n - v_\alpha^n) = P_{\alpha\mathcal{S}R}^*(v_{\theta R}^n - v_{\alpha R}^n) = 0$, so $\theta \in \Delta_n$. ■

The following example illustrates the above ideas for n-optimality and that the subproblems produce distinct solutions.

Example 2.1 Three Subproblems have Distinct Solutions. Consider the deterministic system in Figure 2.1 with $2m+3$ states labeled $\sigma, 0, 1, \ldots, m, 0', 1', \ldots, m'$ where $m \geq 0$. There are two

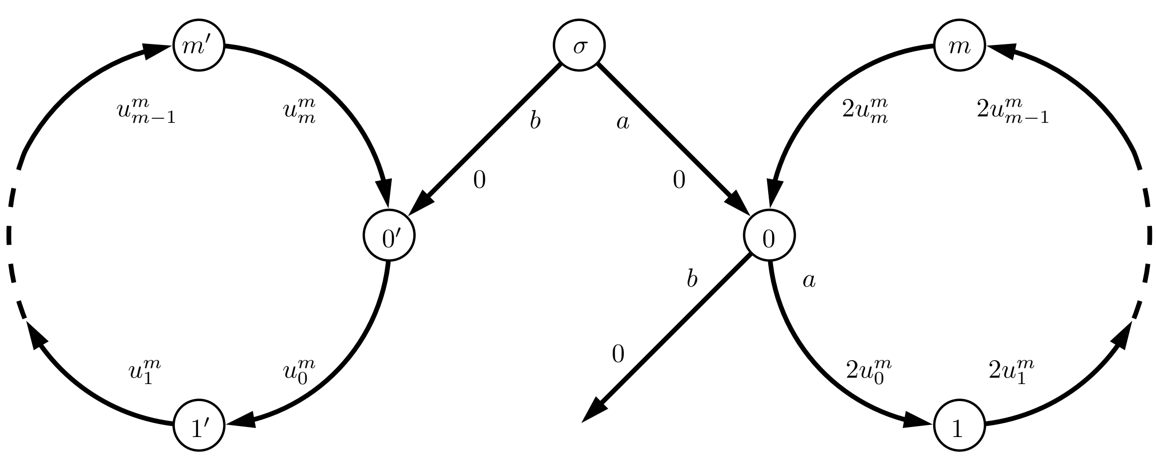


Figure 2.1: Three Subproblems have Distinct Solutions

actions a and b in states σ and 0, and a single action a in every other state. In state σ, actions a and b move the system to states 0 and $0'$ respectively, and each earns zero one-period reward. Action a in states s and s' moves the system to the respective states $s+1$ and $(s+1)'$ modulo $(m+1)$ and earns the respective one-period rewards $2u_s^m$ and u_s^m for $s = 0, \ldots, m$, where $u_s^m \equiv (-1)^s \binom{m}{s}$ for $s = 0, 1, \ldots$. Finally, action b in state 0 earns zero one-period reward and moves the system to the stopped state.

A policy is a pair of actions for states σ and 0. For example, ba is the policy that takes action b in state σ and action a in state 0. Let $n = m - 1$. Then all four policies are $(n-1)$-optimal. If one starts with the policy $\delta = ab$, solving subproblems **(a)**, **(b)** and **(c)** of Theorem 2.1 yields the respective policies $\zeta = bb$, $\eta = ba$ and $\theta = aa$. Also, $v_\delta^n < v_\zeta^n < v_\eta^n < v_\theta^n$ and only θ is n-optimal.

2.4 Efficient Implementation via State-Classification

The subproblems **(a)** and **(c)** are maximum-transient-value problems and the subproblem **(b)** is a maximum-reward-rate problem. One may solve both problems by policy iteration or linear programming. This section first describes two state space partitions and then proceeds to implement both approaches efficiently by partitioning the state space.

2.4.1 Recurrent Classes

As Bather [Bat73, pp. 542–545] shows, it is useful to classify the states of a Markov decision chain as recurrent or transient in a manner that generalizes that for Markov chains. In particular call a state recurrent or transient according as it is recurrent under some policy or transient under every policy. The sets of recurrent and transient states partition the state space.

Bather also gives a recursive procedure for finding the set of recurrent states and a partition thereof that is useful for computations. Ross and Varadarajan [RV91, pp. 197–198] show that the members of this partition are precisely the maximal subsets of recurrent states that form a recurrent class under some randomized stationary Markov policy. For this reason, call each member of the partition a recurrent class.[3] Bather [Bat73, pp. 542–545] and Ross and Varadarajan [RV91, pp. 197–198] give different methods of finding the recurrent classes in $O(S^2 + SA)$ time.

Incidentally, there is a second characterization of the recurrent classes in terms of the (non-randomized stationary Markov) policies considered herein. To describe it, form an (undirected) recurrent-state graph whose nodes are the recurrent states and whose arcs are the pairs (s, t) of states s, t that both belong to a common recurrent class of some policy. Then it is easy to see that the recurrent classes are precisely the node sets of the maximal connected subgraphs (or components) of the recurrent-state graph.

[3] Ross and Varadarajan [RV91] use the term strongly communicating class for what we call a recurrent class.

2.4.2 Communicating Classes

While recurrent classes are useful for solving the maximum-reward-rate subproblem in **(b)** solving the maximum-transient-value subproblems in **(a)** and **(c)** of Theorem 2.1, requires a different (well-known) partition of the state space. It is easiest to explain this in terms of a (directed) graph called the state-graph. The nodes of this graph are the states and the stopped state and the arcs are the ordered pairs of states (s,t), $s \in \mathcal{S}$, for which $p(t|s,a) > 0$ for some $a \in \mathcal{A}_s$ and $t \in \mathcal{S}$ or for which t is the stopped state and $\sum_{t\in\mathcal{S}} p(t|s,a) < 1$. State s is accessible to state t if there is a chain from s to t in the state graph. A state is accessible to itself with a chain of consisting of only that state. A communicating class (or strong component of the state graph) is a maximal subset of mutually accessible states. The communicating classes partition the state space. Tarjan's [Tar72] depth-first search method can be used to find the communicating classes and the communicating-class graph, i.e., the strong component graph of the state-graph, in $O(S + A)$ time. Call a stochastic system communicating if $\mathcal{S}$ forms a communicating class.

2.4.3 Solving the Maximum-Reward-Rate Subproblem

Classifying the states into recurrent classes permits the maximum-reward-rate subproblem in **(b)** of Theorem 2.1 to be divided into smaller subproblems of the same type. The key to doing this is a felicitous choice of the action sets $\underline{\mathcal{E}}^{\,n}_{\zeta s}$. To that end, first form the restricted system by limiting the actions in each state s to those in $\mathcal{E}^n_{\zeta s}$. Next use one of the above algorithms to find the recurrent classes for the restricted system. Now for each state s in some recurrent class C of the restricted system, let $\underline{\mathcal{E}}^{\,n}_{\zeta s}$ be the set of actions in $\mathcal{E}^n_{\zeta s}$ that keep the system in C, i.e., $\underline{\mathcal{E}}^{\,n}_{\zeta s} = \left\{a \in \mathcal{E}^n_{\zeta s} : \sum_{c\in C} p(c|s,a) = 1\right\}$. For each state s that is transient in the restricted system, let $\underline{\mathcal{E}}^{\,n}_{\zeta s} = \{\zeta_s\}$. This definition of $\underline{\mathcal{E}}^{\,n}_{\zeta}$ assures that the required conditions (i) and (ii) hold. Finally, choosing action $a \in \underline{\mathcal{E}}^{\,n}_{\zeta s}$ in state s results in the reward rate $r^{n+1}(s,a) - v^n_{\zeta s}$. The transition rates are the original ones.

Since each recurrent class C of the restricted system has access only to states in C with every policy in $\underline{\mathcal{E}}^{\,n}_{\zeta}$, it suffices to solve the maximum-reward-rate subproblem separately on each such class C. Then one common maximizer $\gamma = \eta$ of both sides of the subproblem **(b)** is formed by letting $\eta_s = \zeta_s$ for each state s that is transient in the restricted system and, for each recurrent class C of the restricted system, letting η_C be a maximum-reward-rate policy for the restriction of the subproblem to C.

To sum up, the above development shows how to divide the maximum-reward-rate subproblem in **(b)** of Theorem 2.1 (with $\underline{\mathcal{E}}^{\,n}_{\zeta}$ defined as above) into a collection of state-disjoint maximum-reward-rate problems. In each of these state-disjoint problems, the system is stochastic and the state-space is a single recurrent class. It is known and easy to see that a stochastic system has a state-space that forms a single recurrent class if and only if the system is communicating. Moreover, much more efficient algorithms are available for communicating stochastic systems than for general ones.

Solving a Maximum-Reward-Rate Problem in a Communicating Stochastic System. Consider a communicating stochastic system. Call a policy unichain if its recurrent states form a single class; naturally, the remaining states are transient. It is known from Denardo [Den70b] that one maximum-reward-rate policy for this problem is unichain. A unichain maximum-reward-rate policy can be found by linear programming or policy iteration.

- **Linear Programming** It is immediate from Denardo [Den70b] that the basic solutions of Manne's [Man60] and de Ghellinck's [dG60] linear program are precisely the stationary distributions of unichain policies. Thus, application of the simplex method, for example, to that linear program iterates within the class of unichain policies until one with maximum reward rate is found. Manne's and de Ghellinck's linear program has only $S+1$ rows and A columns, and so is much smaller than the linear program required for general systems, which has, as discussed above, $2S$ rows and $2A$ columns.

- **Policy Iteration** Haviv and Puterman [HP91] show how to carry out the policy iteration within the class of unichain policies. This is much more efficient than use of general policy iteration for communicating stochastic systems.

2.4.4 Solving the Maximum-Transient-Value Subproblem

One can use the communicating-class graph to solve a maximum-transient-value problem as follows. Let $v^* = (v^*_s)$ be the desired maximum transient value. Choose a communicating class C whose maximum transient values have not been found and that is accessible in one step only to classes for which those values have been found. (Note that the transient value for the stopped state is zero.) Denote by T_C the union of the classes to which C has access in one step. Let the one-period reward associated with each state $s \in C$ and each action a in that state be the sum of its one-period reward and its expected future reward in other classes, i.e., $r(s,a) + \sum_{t \in T_C} p(t|s,a)v^*_t$. Solve the maximum-transient-value problem for that class. The maximum transient values for the class are the desired values v^*_s for each $s \in C$. Repeat this procedure until the maximum transient values for all classes have been found.

2.5 Strongly Polynomial-Time Methods for Unique Transition Systems

In this section we show how to find a strong-present-value optimal policy in strongly polynomial time in a unique transition system. A unique transition system is one in which each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}_s$ determines a unique $t(s,a)$ such that either

- $t(s,a) \in \mathcal{S}$, $p(t(s,a)|s,a) > 0$ and $p(t|s,a) = 0$ for $t \in \mathcal{S} \setminus \{t(s,a)\}$, or
- $t(s,a)$ is the stopped state and $p(t|s,a) = 0$ for each $t \in \mathcal{S}$.

Such systems include standard deterministic dynamic programs. Remember A is the number of state-actions pairs (s,a) with $s \in \mathcal{S}$ and $a \in \mathcal{A}_s$.

A unique transition system induces a state-action graph with gains on the arcs. The nodes are the states and the stopped state. Each arc corresponds to a state-action pair (s,a) with $s \in \mathcal{S}$ and $a \in \mathcal{A}_s$, the tail and head of the arc are s and $t(s,a)$ respectively, and the arc's gain $p(s,a)$ is $p(t(s,a)|s,a)$ if $t(s,a) \in \mathcal{S}$ and $0 < p(s,a) \le 1$ if $t(s,a)$ is the stopped state. Note that this graph may have multiple arcs from one node to another.

A policy induces a subgraph of the state-action graph with each state being the tail of a single arc. Examination of the strong-component graph of the subgraph reveals that the latter is a set of node-disjoint circuit-rooted trees that spans the nodes of the state-action graph. A circuit-rooted tree is a subgraph of the state-action graph whose strong-component graph is a tree with all arcs directed towards a distinguished root node. The root node is a strong component consisting of either a simple circuit or the stopped state, and every other strong component consists of a single node. Figure 2.2 illustrates a circuit-rooted tree with the solid nodes forming the simple circuit.

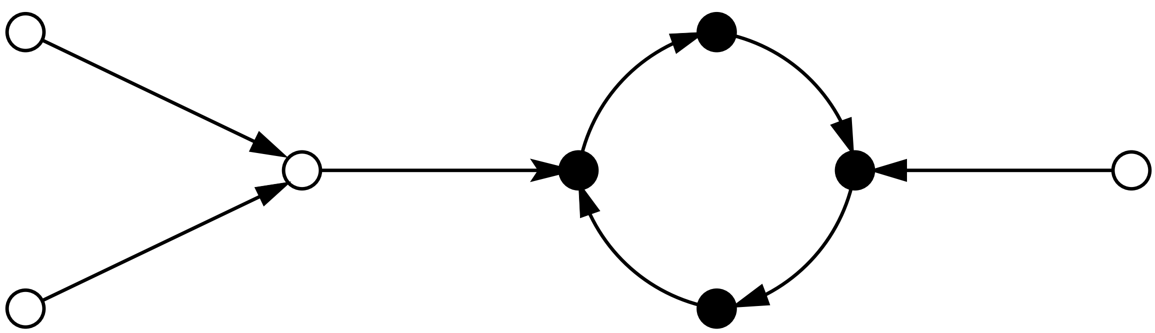


Figure 2.2: Circuit-Rooted Tree

It is possible to classify the states under a policy with the aid of its induced subgraph. Call a simple circuit in the subgraph recurrent if the gains on all arcs in the circuit are one; otherwise call the simple circuit transient. A state in the subgraph is recurrent if it lies in a recurrent simple circuit; otherwise the state is transient. A circuit-rooted tree in the subgraph is transient if its simple circuit, if any, is transient.

Corollary 2.3 Unique Transition Systems. *In a unique transition system, the problem of finding an $(n-2)$-optimal policy, $-1 \le n-2 \le S$, is solvable in*[4]

(a) Undiscounted $O(nS[A \wedge S^2] + A)$ *time if* $p(s,a) = 1$ *for each* $s \in \mathcal{S}$ *and* $a \in \mathcal{A}_s$*;*

[4]The assumption that $n-2 \le S$ is without loss of generality since if $n-2 > S$, a policy is $(n-2)$-optimal if and only if it is S-optimal, or equivalently, strong-present-value optimal.

(b) Discounted *$O(nS^2[A \wedge S^2] + A)$ time if $p(s, a) = \beta < 1$ for each $s \in \mathcal{S}$ and $a \in \mathcal{A}_s$; and*

(c) General *$O(nS^2 A \log S)$ time.*

Proof. For both **(a)** and **(b)**, consider a pair of states (s, t) where t is here possibly the stopped state and t is accessible from s in one step with some action $a \in \mathcal{A}_s$. Then, of course, one can choose an action $a \in \mathcal{A}_s$ that maximizes $r(s, a)$ subject to $t(s, a) = t$ and eliminate the other actions $\alpha \in \mathcal{A}_s$ for which $t(s, \alpha) = t$. This step requires $O(A)$ time and assures that at most $O(A \wedge S^2)$ state-action pairs must be examined, viz., one for each pair (s, t) of states.

(a) Undiscounted Suppose the hypotheses of part **(a)** of Corollary 2.3 hold. Then all circuits in the corresponding collection of circuit-rooted trees are recurrent. Thus each transient policy corresponds to a circuit-rooted tree in which the root component is the stopped state. Then the maximum-transient-value problem may be solved using the Bellman-Ford method [For56, Bel58] for finding a maximum-value rooted tree with root the stopped state. The time to find this tree is $O(S[A \wedge S^2])$. An efficient way to do this is to use state-partitioning as discussed in §2.4.

The maximum-reward-rate problem may be solved by implementing Bather's state-partitioning method as follows. For each class, use Karp's [Kar78] algorithm to find a maximum-reward-rate circuit and then use breadth-first search backwards from the circuit to create a circuit-rooted tree and hence a maximum-reward-rate policy on the class. Finally, solve the system-maximum-reward-rate problem discussed in §2.4 by the Bellman-Ford method. The time to solve both of these problems is $O(S[A \wedge S^2])$.

Thus the time to solve each subproblem **(a)**, **(b)** or **(c)** of Theorem 2.1 is $O(S[A \wedge S^2])$. Since $3n$ such subproblems must be solved, part **(a)** of the Corollary follows immediately.

(b) Discounted Suppose the hypotheses of part **(b)** of the Corollary hold. Then the maximum-transient-value problem may be solved with the aid of state-partitioning. The method entails solving a maximum-transient-value problem on each class. Thus, it suffices to consider the case in which the system consists of a single class. On such a system and for each $\sigma \in \mathcal{S}$, find the maximum-transient-value C_σ over all circuits that start in σ. Then solve an augmented maximum-transient-value problem where in each state $\sigma \in \mathcal{S}$ one also allows stopping and earning the terminal reward C_σ.

One may find a maximum-transient-value circuit that starts in state $\sigma \in \mathcal{S}$ as follows. Calculate the maximum value V_s^N among all N-step chains from $s \in \mathcal{S}$ to σ as follows:

$$V_s^{N+1} = \max_{t(s,a) \in \mathcal{S}} \left[r(s, a) + \beta V_{t(s,a)}^N \right], N \geq 1 \tag{2.15}$$

where $V_s^1 = \max_{t(s,a)=\sigma} r(s, a)$, the maximum being $-\infty$ if the set over which a maximum is taken is empty. Then,

$$C_\sigma = \max_{1 \leq N \leq S} \frac{V_\sigma^N}{1 - \beta^N} = \max_{1 \leq N \leq S} \left[V_\sigma^N + \beta^N C_\sigma \right]. \tag{2.16}$$

Now let U_s^N be the maximum value among all chains of length N or less that begin at $s \in \mathcal{S}$ and earn the terminal reward C_σ if the chain ends in σ. Then for each $s \in \mathcal{S}$,

$$U_s^{N+1} = \max_{a \in \mathcal{A}_s} \left[r(s,a) + \beta U_{t(s,a)}^N \right], N \geq 0 \tag{2.17}$$

where $U_t^N \equiv 0$ if t is the stopped state and $U_s^0 = C_s$. Then $U_s \equiv U_s^S$ is the maximum-transient-value among all policies and satisfies

$$U_s = \max_{a \in \mathcal{A}_s} \left[r(s,a) + \beta U_{t(s,a)} \right].^5$$

Furthermore, any policy δ such that $\delta_s = a$ achieves the maximum on the right-hand side of the above equation for each $s \in \mathcal{S}$ has maximum transient value.

The time to find δ is $O(S^2[A \wedge S^2])$. Thus the time to solve the subproblem **(a)** of Theorem 2.1 is $O(S^2[A \wedge S^2])$. Since the system is transient, any policy that solves subproblem **(a)** also solves subproblems **(b)** and **(c)**, so the latter two are not needed. Since n such subproblems must be solved, part **(b)** of Corollary 2.3 follows immediately.

(c) General Consider the general case of part **(c)** of Corollary 2.3. The maximum-transient value problem can be solved in $O(S^2 A \log S)$ time as follows. Use the method of Hochbaum and Naor [HN94, pp. 1181–1184, 1186] to find the least excessive point of the optimal return operator in $O(S^2 A \log S)$ time.[6] Then from [EV75], the least excessive point is the maximum-transient-value v^*; there is a transient policy δ such that $v^* = r_\delta + P_\delta v^*$; and each such policy has value v^*. As shown in [EV75], one such transient policy $\delta = (\delta_s)$ can be found in $O(SA)$ time as follows. Form a revised state-action graph in two steps. First delete arcs (s,a) in the state-action graph for which $v_s^* \neq r(s,a) + p(s,a) v_t^*(s,a)$. Next, for each state s for which there is an arc (s,a) in the remaining graph for which $p(s,a) < 1$, append an arc $(s,a)'$ from s to the stopped state. Then fan out backwards from the stopped state in the revised state-action graph to form a maximal subtree with all arcs directed towards the stopped state. The subtree spans the states and the stopped state because there is a maximum-value transient policy δ such that $v^* = r_\delta + P_\delta v^*$. Then for each state s, let δ_s be the action associated with the unique arc in the subtree leading out of s. The policy δ is transient since each state is accessible to the stopped state with positive probability under δ.

The maximum-reward-rate problem can also be solved in $O(S^2 A \log S)$ time as follows:

1. remove all arcs (s,a) with $p(s,a) < 1$ to form an undiscounted system;

2. solve the maximum-reward-rate problem on the undiscounted system as outlined in the proof for part **(a)** above;

[5] It is possible to cut the running time to find $U = (U_s)$ using a refinement of Dijkstra's algorithm. However, this improvement does not reduce the order of the leading term of the overall running time.

[6] Their method repeatedly calls the Grapevine Algorithm 1 of Aspvall and Shiloach [AS80, pp. 834–835] to check whether a given number V_s is or is not as large as the maximum transient value in state s in $O(SA)$ time.

3. restore all arcs, augment the system by allowing stopping in each state with the maximum reward-rate from that state in the undiscounted system, and solve the maximum-transient-value problem using the method given above.

Thus, the time to find a maximum-reward-rate policy is $O(S^2 A \log S)$.

Hence, the time to solve each subproblem **(a)**, **(b)** or **(c)** of Theorem 2.1 is $O(S^2 A \log S)$. Since, by Theorem 2.1, $3n$ such subproblems must be solved, part **(c)** of Corollary 2.3 follows. ■

Remark. Hartman and Arguelles [HA99, p. 437], Theorem 17 give an algorithm for finding a maximum-transient-value policy under the hypotheses of **(a)** of Corollary 2.3 and the assumption that the actions in each state consist of choosing the next state to visit. In this event, there are S actions in each state, so $A = S^2$. Under this hypothesis, their method runs in $O(S^4)$ time. The running time of the new method for this case agrees with theirs, but is faster where $A < O(S^2)$.

2.6 Prior Decomposition Methods

The idea of decomposing the problem of finding an n-optimal policy into subproblems related to those proposed here has been explored previously in some special cases. Following is a review of some of this work and it's relationship to the approach presented here.

−1-Optimality (Maximum-Reward-Rate) Problem. Denardo [Den70b] shows how to solve the maximum-reward-rate problem in general by solving Manne's [Man60] and de Ghellinck's [dG60] linear program on a sequence of up to $2n$ "nested" restrictions of the problem. Here, n is the number of recurrent classes under the maximum-reward-rate policy and "nested" means that each restriction has a state-space that is a proper subset of its predecessor.

Bather [Bat73] finds a maximum-reward-rate policy in two steps. First, he separately considers each recurrent class together with those transient classes that feed directly into that class, finding a maximum-reward-rate policy in the subsystem using a simplified version of Howard's policy iteration. Then he uses a maximum-transient-value problem to decide if it is optimal to stay in a recurrent class (using the policy found within it's subsystem), or to move down to another accessible recurrent class. The resulting policy has maximum reward rate for the original system.

0-Optimality Problem. Building on an idea in [Vei66], Denardo [Den70a] shows how to decompose the problem of finding a 0-optimal policy into three subproblems. His first subproblem amounts to finding a maximum-reward-rate policy by policy iteration or linear programming, and so is quite different from subproblem **(a)** for finding a 0-optimal policy. But his second two subproblems are close to subproblems **(b)** and **(c)**, the difference being minor if Blackwell's [Bla62] policy iteration is used to solve these subproblems and being greater if linear programming is used.

***n*-Optimality Problem.** Two prior decompositions for transient and irreducible systems and a decomposition outline for the general case currently exist.

- **Transient Systems** For transient systems, i.e., where every policy is transient, results of Veinott [Vei69, pp. 1638, 1648–1649] imply that it is possible to decompose the problem of finding an n-optimal policy into a sequence of $n+1$ subproblems like subproblem **(c)** except that one can omit the vector v_η^n. This is possible in transient systems because, as he shows, $\Delta_{n-1} = \mathcal{E}_*^{n-1}$ and $\theta \in \mathcal{E}_*^{n-1}$ is n-optimal if and only if $\max_{\gamma \in \mathcal{E}_*^{n-1}} g_{\gamma\theta}^n = 0$, i.e., $v^n = v_\theta^n$ is the unique fixed point of (2.12). This result also follows from Theorem 2.1 on observing that a policy θ that solves subproblem **(c)** for n also solves subproblems **(a)** and **(b)** for $n+1$.

- **Irreducible Systems** For irreducible systems, i.e., where every policy is irreducible, Avrachenkov and Altman [AA98] show how to decompose the problem of finding an n-optimal policy into a sequence of $n+2$ subproblems like subproblem **(b)**. This result also follows from Theorem 2.1 on observing that a policy η that solves subproblem **(b)** for n also solves subproblem **(c)** for n and subproblem **(a)** for $n+1$.

2.7 Contributions

This chapter presents a new three-step decomposition for finding strong-present-value optimal policies in substochastic systems. None of the three steps require a specific methodology, so the entire decomposition is independent of the solution methods used at each step. If linear programming is used to solve each step of the decomposition, the entire process runs in polynomial time with respect to the size of the dynamic program. Also, this chapter shows how to use two different state space partitions (recurrent and communicating classes) to solve each step efficiently (again, no particular solution method is specified). Finally, this chapter defines unique transition systems and shows that the decomposition presented here may solve in strongly polynomial time (with respect to the size of the dynamic program) for these systems, with the exact running time depending on the properties of the system (and the specialized methods used to solve each step of the decomposition).

Acknowledgement The work contained within this chapter is joint work with Prof. Arthur F. Veinott, Jr.

Chapter 3

Numerically Stable Present-Value Expansion

3.1 Introduction

One of the most important steps in implementing any method is ensuring the method remains valid under finite precision. Both Miller and Veinott's [MV69] and Veinott's [Vei69] policy improvement for substochastic systems and Rothblum's [Rot75] extended policy improvement for normalized systems require the computation of Laurent expansion coefficients via a set of linear equations. If these coefficients are not computed to a high degree of accuracy, then these policy improvement methods may be invalidated. This chapter presents a method for solving the linear equations (thus computing the coefficients) for systems with substochastic classes (a subset of normalized systems).

Section 3.2 presents the DP formulation, describes the systems considered in this chapter, and gives the modified Laurent expansion and linear equations for these systems. Then, Section 3.3 shows how to use a the classes of a policy to decompose the problem of solving the linear equations for a policy into many problems that solve the linear equations within a substochastic, irreducible class. Section 3.4 shows how to perform numerically stable computations to solve these linear equations within a substochastic, irreducible class. This provides the necessary building block to compute the coefficients for the entire system in a numerically stable way. Finally, the question of numerical stability is addressed in detail in §3.5 and the contributions of this chapter are summarized in §3.6.

3.2 Preliminaries

3.2.1 Formulation

Consider a *general system* observed in periods $1, 2, \ldots$. The system exists in a finite set $\mathcal{S}$ of S states. In each state $s \in \mathcal{S}$ the system takes one of a finite set $\mathcal{A}_s$ of actions. The number of state-action pairs is given by A. Taking action $a \in \mathcal{A}_s$ from $s \in \mathcal{S}$ earns *reward* $r(s,a)$ and causes a *transition* to state $t \in \mathcal{S}$ with rate $p(t|s,a)$.

Each (stationary) policy $\delta \in \Delta$ is a function that assigns a unique action $\delta_s \in \mathcal{A}_s$ to each $s \in \mathcal{S}$, and induces a single-period S-column *reward vector* $r_\delta \equiv (r(s,\delta_s))$ and an $S \times S$ *transition matrix* $P_\delta \equiv (p(t|s,\delta_s))$.

3.2.2 Systems with Substochastic Classes

The matrix P_δ defines a Markov chain, complete with recurrent and transient classes under δ. If every class $\mathcal{C}$ (under δ) has $0 \leq p(t|s,a) \leq 1$ and $\sum_{t\in\mathcal{C}} p(t|s,a) \leq 1,\ s,t \in \mathcal{C},\ a \in \mathcal{A}_s$, then δ has substochastic classes. Also, since the blocks of P_δ corresponding to these classes lie on the diagonal and have spectral radius not exceeding one, P_δ has spectral radius not exceeding one. If every (stationary) policy has substochastic classes, then the system has substochastic classes. Also, since the spectral radius of the transition matrix for every (stationary) policy does not exceed one, the system is normalized. Hereafter, this chapter only considers system with substochastic classes.

Blackwell's [Bla62] existence theorem holds for normalized systems, so it suffices to restrict attention to stationary policies throughout this chapter. Rothblum [Rot75] shows how to extend the policy improvements from Miller and Veinott [MV69] and Veinott [Vei69] (for substochastic systems) to normalized systems. The only differences arise in the Laurent expansion and the linear equations needed to find the coefficients of this expansion.

3.2.3 Laurent Expansion

For each policy δ, let the *degree* of δ be the smallest nonnegative integer $d_\delta \equiv i$ such that Q_δ^i and Q_δ^{i+1} have the same null space (where $Q_\delta \equiv P_\delta - I$). Let the *system degree* $d \equiv \max_{\delta\in\Delta} d_\delta$. For normalized systems, the Laurent expansion becomes

$$V_\delta^\rho = \sum_{n=-d}^{\infty} \rho^n v_\delta^n. \tag{3.1}$$

If $V^{n+d} \equiv \left(v^{-d}, \ldots, v^{n+d}\right)$ is a solution of

$$r_\delta^j + Q_\delta v^j = v^{j-1}, j = -d, \ldots, n+d, \tag{3.2}$$

where $r^0(s,a) \equiv r(s,a)$, $r^j(s,a) \equiv 0$ for $j \neq 0$, $r^j_\delta \equiv \left(r^j(s,\delta_s)\right)$ and $v_\delta^{-d-1} = 0$, then $V^n \equiv \left(v_\delta^{-d}, \ldots, v_\delta^n\right) = \left(v^{-d}, \ldots, v^n\right)$. The variables $v^{n+1}, \ldots, v^{n+d}$ are not uniquely determined.

The remainder of this chapter presents a numerically stable method for finding V_δ^n for a given $\delta \in \Delta$ and $-d \leq n$. More generally, the method finds a solution of

$$c^j + Q_\delta v^j = v^{j-1},\ j = m+1, \ldots, n+d \tag{3.3}$$

given $\delta \in \Delta$, v^m and c^j, $j = m+1, \ldots, n+d$. Again, only v^j, $j = m+1, \ldots, n$ are uniquely determined.

3.3 Using the Class Structure

Given a policy δ (in a system with substochastic classes), the transition matrix P_δ not only defines (recurrent and transient) classes, but also induces a dependence partial ordering amongst the classes. In substochastic systems, the only classes that don't depend on any other class are recurrent. All other classes are transient. Veinott solves for the Laurent coefficients by solving the linear equations within each recurrent class separately (accounting for the singularity present in these classes) and using these coefficients to solve for the rest of the system (that is transient). However, in systems with substochastic classes, recurrent classes may depend on other (recurrent or transient) classes.

Let $\mathcal{D}_0$ be the set of classes that are *independent*, i.e., don't depend on any other classes. Let $\mathcal{D}_i$ be the set of classes that depend only on classes in $\cup_{j=0}^{i-1}\mathcal{D}_j$ and let $\mathcal{S}_i \equiv \left\{s \in \mathcal{C} \in \cup_{j=0}^{i-1}\mathcal{D}_j\right\}$.

For each class $\mathcal{C}$ in $\mathcal{D}_i$, consider (3.2) for $s \in \mathcal{C}$. The dependence partial ordering shows that $p(t|s,\delta_s) > 0$ only for t in $\mathcal{C} \cup \mathcal{S}_i$. If one has already calculated $V_t^{n+d} \equiv \left(V_{\delta t}^n, v_t^{n+1}, \ldots, v_t^{n+d}\right)$ for $t \in \mathcal{S}_i$, then solving (3.2) within $\mathcal{C}$ reduces to solving (3.3) within $\mathcal{C}$, with $m \equiv -d-1$, $v_\delta^{-d-1} \equiv 0$ and $c_s^j \equiv r^j(s,\delta_s) + \sum_{t \in \mathcal{S}_i} p(t|s,\delta_s) v_t^j$ for $s \in \mathcal{C}$. By solving (3.3) within the classes in $\mathcal{D}_i$, $i = 0, 1, \ldots,$ in order one may solve (3.2) for the entire system efficiently. Also, this is done by solving (3.3) within substochastic, irreducible classes.

3.4 Substochastic, Irreducible Classes

This section presents a method for solving the linear system (3.3) within a single (substochastic, irreducible) class. Throughout, the notation for system parameters denotes those same parameters restricted to the class, e.g., $\mathcal{S}$ refers to the states within the class, P_δ refers to the transition matrix restricted to the states in the class, etc.

Each class may be either transient or recurrent. In a transient class, $\lim_{N\to\infty} P_\delta^N = 0$, so $Q_\delta^{-1} \equiv (P_\delta - I)^{-1} = -\left(I + P_\delta + P_\delta^2 + \cdots\right)$ is well-defined (i.e., Q_δ is non-singular). If the class is recurrent, then it must be *stochastic*, i.e., $\sum_{t \in \mathcal{S}} p(t|s,\delta_s) = 1$ for every $s \in \mathcal{S}$. Then the rows of Q_δ sum to zero, so Q_δ is singular. Since the class is irreducible, eliminating any (single) state s

from the class causes the remainder to become transient. This is equivalent to removing the row corresponding to the state-action pair (s, δ_s) and the column corresponding to s from P_δ. Removing this row and column from Q_δ causes it to become non-singular. Therefore, Q_δ has rank $S-1$, i.e., it is *rank-deficient by one*.

3.4.1 Rank-revealing LU factors

Given a (substochastic, irreducible) class, one may deduce if it is transient or recurrent by means of a *rank-revealing LU (RRLU) factorization* of Q_δ. Moreover, regardless of the nature of the class, the LU factors may still be used to solve (3.3).

The RRLU takes the form

$$T_1 Q_\delta T_2^T \equiv T_1 \begin{pmatrix} \hat{Q} & \hat{q} \\ q^T & \varphi \end{pmatrix} T_2^T = \begin{pmatrix} L & 0 \\ l^T & 1 \end{pmatrix} \begin{pmatrix} U & u \\ 0 & \varepsilon \end{pmatrix},$$

where T_1 and T_2 are permutations, $\hat{Q} = LU$ is an $(S-1)\times(S-1)$ non-singular matrix, q^T and l^T are $(S-1)$-row vectors, $\hat{q}$ and u are $(S-1)$-column vectors, and ε is a valuable indicator if the permutations are chosen correctly (according to the Threshold Complete Pivoting strategy described in Chapter 4). If $|\varepsilon|$ is suitably large Q_δ is taken as full rank, but if $|\varepsilon| = O(\epsilon)$, where ϵ is the machine precision, Q_δ is regarded as singular. In particular, it is rank-deficient by one.

For simplicity, assume the permutations T_1 and T_2 are the identity and write the LU factorization as

$$Q_\delta \equiv \begin{pmatrix} \hat{Q} & \hat{q} \\ q^T & \varphi \end{pmatrix} = \begin{pmatrix} L & 0 \\ l^T & 1 \end{pmatrix} \begin{pmatrix} U & u \\ 0 & \varepsilon \end{pmatrix}.$$

Regardless of the nature of the class, the LU factors may be used to solve (3.3) because the system is consistent even if Q_δ is singular.

3.4.2 Transient classes

If the RRLU factorization shows that Q_δ has full rank, then the linear system (3.3), equivalently

$$\begin{bmatrix} Q_\delta & & & \\ -I & Q_\delta & & \\ & \ddots & \ddots & \\ & & -I & Q_\delta \end{bmatrix} \begin{bmatrix} v^{m+1} \\ v^{m+2} \\ \vdots \\ v^{n+d} \end{bmatrix} = \begin{bmatrix} v^m - c^{m+1} \\ -c^{m+2} \\ \vdots \\ -c^{n+d} \end{bmatrix}, \tag{3.4}$$

is nonsingular and block triangular. It may therefore be solved sequentially by *block forward substitution*:

$$Q_\delta v^j = v^{j-1} - c^j \tag{3.5}$$

in order for $j = m+1, \ldots, n+d$ (using the full LU factors of Q_δ).

3.4.3 Recurrent classes

If the RRLU factorization shows that Q_δ is rank-deficient by one, then the linear system (3.3) may be represented in matrix form

$$\begin{bmatrix} \hat{Q} & \hat{q} & & & & & & \\ q^T & \varphi & & & & & & \\ -I & & \hat{Q} & \hat{q} & & & & \\ & -1 & q^T & \varphi & & & & \\ & & \ddots & & \ddots & & & \\ & & & & -I & & \hat{Q} & \hat{q} \\ & & & & & -1 & q^T & \varphi \end{bmatrix} \begin{bmatrix} \hat{v}^{m+1} \\ v_S^{m+1} \\ \hat{v}^{m+2} \\ v_S^{m+2} \\ \vdots \\ \hat{v}^{n+d} \\ v_S^{n+d} \end{bmatrix} = \begin{bmatrix} \hat{v}^m - \hat{c}^{m+1} \\ v_S^m - c_S^{m+1} \\ -\hat{c}^{m+2} \\ -c_S^{m+2} \\ \vdots \\ -\hat{c}^{n+d} \\ -c_S^{n+d} \end{bmatrix}, \tag{3.6}$$

where $\hat{v}^j$ and $\hat{c}^j$ represent the first $S-1$ elements of v^j and c^j respectively. Rearranging (3.6) gives

$$\left[\begin{array}{cccc|cccc} \hat{Q} & & & & \hat{q} & & & \\ -I & \hat{Q} & & & & \hat{q} & & \\ & \ddots & \ddots & & & & \ddots & \\ & & -I & \hat{Q} & & & & \hat{q} \\ \hline q^T & & & & \varphi & & & \\ & q^T & & & -1 & \varphi & & \\ & & \ddots & & & \ddots & \ddots & \\ & & & q^T & & & -1 & \varphi \end{array}\right] \left[\begin{array}{c} \hat{v}^{m+1} \\ \hat{v}^{m+2} \\ \vdots \\ \hat{v}^{n+d} \\ \hline v_S^{m+1} \\ v_S^{m+2} \\ \vdots \\ v_S^{n+d} \end{array}\right] = \left[\begin{array}{c} \hat{v}^m - \hat{c}^{m+1} \\ -\hat{c}^{m+2} \\ \vdots \\ -\hat{c}^{n+d} \\ \hline v_S^m - c_S^{m+1} \\ -c_S^{m+2} \\ \vdots \\ -c_S^{n+d} \end{array}\right].$$

Since Q_δ is rank-deficient by one, the last row of the system

$$Q_\delta v^{m+1} = v^m - c^{m+1},$$

i.e., $q^T \hat{v}^{m+1} + \varphi v_S^{m+1} = v_S^m - c_S^{m+1}$, is redundant and may be omitted. Also, since v^{n+d} is not determined uniquely, one may assign $v_S^{n+d} \equiv 0$ and eliminate the column corresponding to this variable. The remaining system

$$\left[\begin{array}{ccccc|cccc} \hat{Q} & & & & & \hat{q} & & & \\ -I & \hat{Q} & & & & & \hat{q} & & \\ & \ddots & \ddots & & & & & \ddots & \\ & & -I & \hat{Q} & & & & & \hat{q} \\ & & & -I & \hat{Q} & & & & \\ \hline & q^T & & & & -1 & \varphi & & \\ & & \ddots & & & & \ddots & \ddots & \\ & & & q^T & & & & -1 & \varphi \\ & & & & q^T & & & & -1 \end{array}\right] \left[\begin{array}{c} \hat{v}^{m+1} \\ \hat{v}^{m+2} \\ \vdots \\ \hat{v}^{n+d-1} \\ \hat{v}^{n+d} \\ \hline v_S^{m+1} \\ v_S^{m+2} \\ \vdots \\ v_S^{n+d-1} \end{array}\right] = \left[\begin{array}{c} \hat{v}^m - \hat{c}^{m+1} \\ -\hat{c}^{m+2} \\ \vdots \\ -\hat{c}^{n+d-1} \\ -\hat{c}^{n+d} \\ \hline -c_S^{m+2} \\ \vdots \\ -c_S^{n+d-1} \\ -c_S^{n+d} \end{array}\right] \tag{3.6'}$$

has a unique solution (as it has full rank). Label the various components shown in (3.6') as

$$\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right] \left[\begin{array}{c} \hat{v} \\ \hline v_S \end{array}\right] = \left[\begin{array}{c} \hat{c} \\ \hline c_S \end{array}\right]. \tag{3.7}$$

Since $\hat{Q}$ is nonsingular (and $\hat{Q} = LU$ has already been found) it is easy to solve a system $Ax = b$ sequentially. Hence a block LU factorization,

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} A & \\ C & I \end{bmatrix} \begin{bmatrix} I & Y \\ & Z \end{bmatrix},$$

solves the full system efficiently. First, solve $AY = B$ and calculate the *Schur complement* $Z = D - CY$, then use block-forward and block-backward substitution, i.e., solve

$$\begin{bmatrix} A & \\ C & I \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_S \end{bmatrix} = \begin{bmatrix} \hat{c} \\ c_S \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} I & Y \\ & Z \end{bmatrix} \begin{bmatrix} \hat{v} \\ v_S \end{bmatrix} = \begin{bmatrix} \hat{x} \\ x_S \end{bmatrix},$$

to solve (3.6'). The remainder of this section breaks down each step of the process.

Solving $AY = B$. Expanding A and B into matrix form gives

$$\begin{bmatrix} \hat{Q} & & & & \\ -I & \hat{Q} & & & \\ & \ddots & \ddots & & \\ & & -I & \hat{Q} & \\ & & & -I & \hat{Q} \end{bmatrix} Y = \begin{bmatrix} \hat{q} & & & \\ & \hat{q} & & \\ & & \ddots & \\ & & & \hat{q} \\ & & & \end{bmatrix}.$$

Therefore, the first column of Y solves the following block-diagonal system:

$$\begin{bmatrix} \hat{Q} & & & & \\ -I & \hat{Q} & & & \\ & \ddots & \ddots & & \\ & & -I & \hat{Q} & \\ & & & -I & \hat{Q} \end{bmatrix} \begin{bmatrix} y^{m+1} \\ y^{m+2} \\ \vdots \\ y^{n+d-1} \\ y^{n+d} \end{bmatrix} = \begin{bmatrix} \hat{q} \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}.$$

One may solve the system sequentially (cf. §3.4.2), i.e., $LUy_j = y_{j-1}$ in order for $j = m+1, \ldots, n+d$ (where $y_m \equiv \hat{q}$). The blocks of the first column of Y give all elements of the following *Toeplitz* matrix:

$$Y = \begin{bmatrix} y^{m+1} & & & \\ y^{m+2} & y^{m+1} & & \\ \vdots & y^{m+2} & \ddots & \\ y^{n+d-1} & \vdots & \ddots & y^{m+1} \\ y^{n+d} & y^{n+d-1} & & y^{m+2} \end{bmatrix}.$$

Forming $Z = D - CY$. Since C is block-diagonal and Y is block-upper triangular, the product CY has the form

$$CY = \begin{bmatrix} q^T y^{m+2} & q^T y^{m+1} & & \\ q^T y^{m+3} & q^T y^{m+2} & \ddots & \\ \vdots & \vdots & \ddots & q^T y^{m+1} \\ q^T y^{n+d} & q^T y^{n+d-1} & & q^T y^{m+2} \end{bmatrix},$$

giving

$$\begin{aligned} Z &= \begin{bmatrix} -1 & \varphi & & \\ & -1 & \ddots & \\ & & \ddots & \varphi \\ & & & -1 \end{bmatrix} - \begin{bmatrix} q^T y^{m+2} & q^T y^{m+1} & & \\ q^T y^{m+3} & q^T y^{m+2} & \ddots & \\ \vdots & \vdots & \ddots & q^T y^{m+1} \\ q^T y^{n+d} & q^T y^{n+d-1} & & q^T y^{m+2} \end{bmatrix} \\ &= -\begin{bmatrix} 1 + q^T y^{m+2} & q^T y^{m+1} - \varphi & & \\ q^T y^{m+3} & 1 + q^T y^{m+2} & \ddots & \\ \vdots & \vdots & \ddots & q^T y^{m+1} - \varphi \\ q^T y^{n+d} & q^T y^{n+d-1} & & 1 + q^T y^{m+2} \end{bmatrix}. \end{aligned}$$

However, $\hat{Q}y^{m+1} = \hat{q}$, so $q^T y^{m+1} = \varphi$ (because Q_δ is rank-deficient by one), so Z is both lower triangular and Toeplitz:

$$Z = -\begin{bmatrix} 1+q^T y^{m+2} & & & \\ q^T y^{m+3} & 1+q^T y^{m+2} & & \\ \vdots & \vdots & \ddots & \\ q^T y^{n+d} & q^T y^{n+d-1} & & 1+q^T y^{m+2} \end{bmatrix}.$$

Block-forward substitution. The solution $\hat{x}$ of $A\hat{x} = \hat{c}$ is found sequentially. Then x_S may be calculated directly from $x_S = c_S - C\hat{x}$.

Block-backward substitution. If Z is large, the solution of $Zv_S = x_S$ may be found using special methods for (lower) triangular Toeplitz systems [Tre87]. Otherwise (Z is of moderate size) and ordinary forward substitution suffices to find v_S. Then, $\hat{v}$ is obtained directly from $\hat{v} = \hat{x} - Yv_S$.

3.5 Numerical Stability

Regardless of whether a class is transient or recurrent, one will always solve a block-lower triangular system sequentially (block-forward substitution) using a nonsingular matrix Q ($\equiv Q_\delta$ for transient classes, $\equiv \hat{Q}$ for recurrent classes). Since the block-forward substitution is equivalent to using $(Q)^{j-m}$, $j = m+1, \ldots, n+d$, any errors in solving with Q will grow exponentially. If Q is ill-conditioned, one may reduce this effect by keeping the *interval* $n+d-m$ in (3.3) small. Implementing policy improvement as suggested by Veinott [Vei69], i.e., finding m-optimal policies for $m = -d, \ldots, n$ sequentially, gives V_δ^{m-1} throughout and one only searches for $m, \ldots, m+d$-improvements (again in order). Finding an $(m+d)$-improvement requires V_δ^{m+d}, so the interval for (3.3) is (at most) $m+2d-(m-1) = 2d+1$. Therefore, the smaller the degree of the system, the more reliable calculations become.

The RRLU is essential in this method. If a class is recurrent, but the LU fails to identify the singularity in Q, then Q will be extremely ill conditioned and computational errors may become prominent.

For recurrent classes, the methods works with the block LU factorization (3.7). The factorization is stable as long as A is not almost singular and the elements of either B or C (or both) are not much larger than the biggest element of A.

The Schur complement Z is lower triangular with the constant diagonal element $\phi_Z \equiv 1+q^T y^{m+2}$. The condition of Z will be reasonable if ϕ_Z is not close to zero and not significantly smaller than the off-diagonal elements $q^T y^{m+3}, \cdots, q^T y^{n+d}$.

3.6 Contributions

Given a policy in a system with substochastic classes, this chapter presents a new method for finding Laurent expansion coefficients (for the present value of the policy). A rank-revealing LU factorization indicates if each (communicating) class is transient or recurrent. For transient classes, the LU factors provide a numerically stable method for finding the coefficients. For recurrent classes, this chapter gives a matrix decomposition based on well defined submatrices of the LU factors to find the coefficients. Finally, this chapter discusses where numerical instability may arise during computation of the coefficients.

Acknowledgement The work contained within this chapter is joint work with Prof. Michael A. Saunders.

Chapter 4

On a Rank-Revealing Sparse LU Factorization

4.1 Introduction

This chapter presents a rank-revealing LU (RRLU) factorization for general sparse matrices (which may be square or rectangular). Although dense RRLU factorizations exist [Cha84, HLY92], as do sparse LU factorizations ([DR96, DEG$^+$99] and many others), no previous sparse LU factorizations have focused on rank-revealing properties. The method presented here is implemented as a new option within LUSOL, the sparse LU factorization described by Gill, Murray, Saunders, and Wright [GMSW87]. LUSOL uses a Markowitz pivot strategy to maintain sparsity, along with *threshold partial pivoting* (TPP) to preserve stability. The new option implements *threshold complete pivoting* (TCP). While requiring relatively few changes to the code, it provides significant rewards. In particular, the stricter stability test is guaranteed to move the smaller pivots (if any) to the end of the factorization. This is sufficient to reveal the rank of most matrices except for the best-known pathological example. The TCP factorization also tends to preserve sparsity well. The main challenge is to stay as near as possible to the linear running time that is usually achieved by the conventional TPP factorization.

Section 4.2 describes various forms of LU factorization, including the original TPP factorization and the modifications made to LUSOL to implement the TCP pivoting strategy. Section 4.3 illustrates the rank-revealing property of the new factorization on some small classical test cases. Next, §4.4 describes experiments on many sparse examples from a dynamic programming application. The complexity appears to be polynomial but less than quadratic in the number of nonzeros in the LU factors. Section 4.5 describes the usefulness of the new factorization within optimization codes. Finally, §4.6 summarizes the contributions within this chapter.

4.2 LU Factorization

The LU factors of a matrix A may be written as $A = LU = \sum_k l_k u_k^T$, where l_k and u_k^T are the columns of L and the rows of U. With $A^{(1)} = A$, the kth factorization step produces

$$A^{(k+1)} = A^{(k)} - l_k u_k^T,$$

in which some nonzero element $A_{ij}^{(k)}$ is chosen as "pivot", and the associated row i and column j become zero. (Thus, $A^{(k+1)}$ contains k empty rows and columns.) For simplicity k is omitted and each step is written as

$$l = A_{.j}/A_{ij}, \qquad u^T = A_{i.}, \qquad A \leftarrow A - lu^T,$$

The resulting L and U are permuted triangles. In pivot order, the diagonals of L are 1 and the pivots A_{ij} become the diagonals of U.

The process is *stable* if the elements of A do not grow large at any stage, and the factors tend to be *sparse* if the rank-one updates lu^T are formed from sparse pivot rows and columns. In practice, one attempts to control element growth by bounding the elements of L. The following terms are needed (where "A" still means $A^{(k)}$):

FactorTol A stability tolerance for controlling the magnitude of $|L_{ij}|$. Pivots are chosen so that $1 \le \|l\|_\infty \le \texttt{FactorTol} \le 100$ (say).

DropTol A tolerance for ignoring negligible entries. Updated elements are treated as zero if $|A_{ij}| < \texttt{DropTol}$. Since the concern is rank estimation, a suitable value is $\texttt{DropTol} = \epsilon\|A\|$, where $\epsilon \approx 2.2 \times 10^{-16}$ is the precision of most of today's machines.

Markowitz strategy A method for choosing sparse updates [Mar57]. A potential pivot A_{ij} is desirable if the merit function $M_{ij} \equiv (r_i - 1)(c_j - 1)$ is low, where r_i and c_j are the number of nonzeros in the associated row and column of the current matrix A. (M_{ij} bounds the number of new nonzeros created by lu^T.)

Threshold Partial Pivoting A strategy for balancing stability and sparsity. A pivot chosen from the remaining columns of A should not be too small compared to other nonzeros in its own column.

Threshold Complete Pivoting A strategy with stricter stability test. A pivot chosen from the remaining columns of A should not be too small compared to other nonzeros in *all* columns.

Modified columns Those that change during the lu^T update (excluding the pivot column and columns whose element of u is zero).

nnz(A) The number of nonzeros in the sparse matrix A.

4.2.1 Dense Partial and Complete Pivoting

For a dense $n \times n$ matrix, LU factorization requires about $\frac{1}{3}n^3$ arithmetic operations. Partial pivoting is usually employed with `FactorTol` $= 1$ (since sparsity is not an issue). Finding the largest element in each pivot column requires $O(n^2)$ comparison operations in total (negligible compared to the arithmetic operations).

Complete pivoting is known to be numerically preferable, as it is even more likely to prevent element growth. Finding the largest element in all remaining rows and columns requires about $\frac{1}{3}n^3$ comparisons, almost doubling the cost of dense LU factorization. (Note that in the sparse case, one would be delighted if complete pivoting cost only twice as much as sparse partial pivoting.)

Businger [Bus71] has shown how to avoid most of the extra cost by switching from partial to complete pivoting only in rare circumstances: when the elements of u_k become too large, not counting the pivots. (This involves monitoring the off-diagonal elements of the current rows of U.) For dense matrices only $O(n^2)$ comparisons are required. Businger's approach also seems suitable in the sparse case (to ensure stability). It will be considered for future implementation.

4.2.2 Stability and Rank Detection

By definition, a *stable* method computes LU factors for any matrix, regardless of condition or singularity, with the relation $A = LU$ holding to high accuracy in all cases. If A is ill-conditioned, at least one of L or U must be also. For example, consider $n \times n$ matrices of the form

$$B_n = \begin{pmatrix} 0.1 & 1 & & & \\ & 0.1 & 1 & & \\ & & \cdot & \cdot & \\ & & & 0.1 & 1 \\ & & & & 0.1 \end{pmatrix},$$

for which $\mathrm{cond}(B_n) \approx 10^n$ with only one small singular value. For $n \geq 15$, B_n is essentially singular and $\mathrm{rank}(B_n) = n - 1$. Partial pivoting exhibits perfect stability by returning $L = I$ and $U = B_n$. Complete pivoting cannot be more stable, but by disallowing the diagonal 0.1 pivots, it ultimately returns a U with one small diagonal U_{nn}, clearly indicating the rank.

Note that MATLAB's `condest` [HT00] correctly estimates $\mathrm{cond}(B_{15}) \approx 10^{15}$ and returns a vector v for which $B_{15}v = O(\epsilon)$. However, the elements of v are smoothly graded: $v = [0.9,\ -0.09,\ 0.009,\ \ldots,\ -9 \times 10^{-14},\ 9 \times 10^{-15}]^T$, giving no indication of the source or degree of singularity. For the applications discussed later, the requirements are these:

- An RRLU factorization should be *stable* in the sense that $A = LU$ holds to high accuracy, and *sparse* whenever A is sparse.
- The factors should indicate which columns of A could be removed (as few as possible) to improve the condition significantly.

4.2.3 Threshold Partial Pivoting (TPP)

Threshold pivoting methods balance stability and sparsity by using FactorTol > 1. For TPP the Markowitz strategy suggests sparse pivots A_{ij} and the stability test requires

$$|A_{ij}| \geq A_{j\,\max} \,/\, \texttt{FactorTol}, \tag{4.1}$$

where $A_{j\,\max}$ is the largest element in column j. This is the strategy used by traditional sparse LU packages, including LA05, MA28, MA48, Y12M, LUSOL and MOPS [Rei82, Duf77, DR96, ZWS81, GMSW87, SS90] (some of them oriented by rows rather than columns).

Typically, $\mathrm{nnz}(L+U)$ is 1–3 times as many as $\mathrm{nnz}(A)$ for the original A (and rarely as much as 10 times). Since A tends to have only 1–10 entries per column and the average Markowitz count M_{ij} tends to be low, the total storage and work required are both typically $O(n)$. Various implementation strategies have been proposed to achieve this efficiency. In particular:

- S1: Zlatev [Zla80] searches only a few of the sparsest rows and columns of the updated A for suitable pivots.
- S2: Suhl and Suhl [SS90] store the largest element of each column of the updated A at the beginning of the column's data structure, to facilitate the stability test.

Both strategies are used in the current version of LUSOL. Strategy S2 is especially important for our rank-revealing implementation. Note that the largest element must be found in each *modified column* (defined above), but the total searching involved in Strategy S2 tends to be linear in $\mathrm{nnz}(L+U)$, as desired.

4.2.4 Threshold Rook Pivoting (TRP)

A stricter stability test would be

$$|A_{ij}| \geq A_{j\,\max} \,/\, \texttt{FactorTol} \quad \text{and} \quad |A_{ij}| \geq A_{i\,\max} \,/\, \texttt{FactorTol}, \tag{4.2}$$

where $A_{i\,\max}$ is the largest element row i. This *threshold rook pivoting* (TRP) strategy has been implemented by Gupta [Gup00] in the WSMP package for judicious use when the preferred TPP strategy fails. For the dense case with FactorTol $= 1$, Foster [Fos97] has shown that rook pivoting prevents exponential growth in the size of $|A_{ij}|$, and suggests that it has rank-revealing properties. The threshold form therefore warrants future study. In the LUSOL context, TRP would complicate the Markowitz strategy (perhaps requiring the nonzeros to be stored by rows as well as columns), but it may involve less searching overall than the TCP strategy described next.

4.2.5 Threshold Complete Pivoting (TCP)

This is the new option implemented in LUSOL. The stability test is

$$|A_{ij}| \geq A_{\max} \,/\, \texttt{FactorTol}, \tag{4.3}$$

where $A_{\max}$ is the largest element in the current A. While the changes in the pivot selection scheme are conceptually minor, they allow the rank of the matrix to be accurately revealed by moving small elements to the later pivot positions. The Markowitz strategy becomes slightly *simpler*, but may continue longer before an acceptable pivot is found. There are two significant costs:

- Decreased sparsity in L and U compared to TPP.
- The need to maintain $A_{\max}$ at each stage.

The first cost seems inevitable, but Strategy S2 helps with the second. With the largest element of each column known, $A_{\max}$ could be found at stage k by searching $n-k$ elements. However, this approach entails $O(n^2)$ comparisons in total. To economize, after each elimination we find

$$A_{\text{mod}} \equiv \text{the largest element in the modified columns}$$

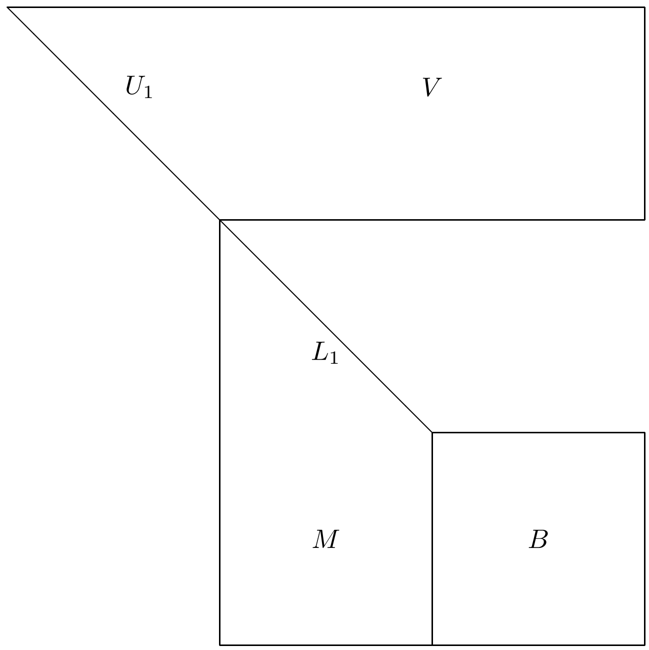
and then consider four cases. Suppose column $j_{\max}$ contains $A_{\max}$ *before* the elimination.

1. If column $j_{\max}$ was modified (or deleted as pivot column),
   1a. if $A_{\max} \leq A_{\text{mod}}$ then $A_{\max} \leftarrow A_{\text{mod}}$
   1b. else search all columns for a new $A_{\max}$
2. else
   2a. if $A_{\max} < A_{\text{mod}}$ then $A_{\max} \leftarrow A_{\text{mod}}$
   2b. else $A_{\max}$ remains the same.

Only case 1b requires a search of $n-k$ elements. The complexity now depends on how often this case arises, i.e., *how often does $A_{\max}$ decrease?* For matrices with many nonzeros of equal magnitude, it may not be often. Otherwise, it may be rather often, especially when `FactorTol` is rather low. For the sparse test matrices of §4.4, the total work appears to be less than $O(n^2)$ but significantly more than $O(n)$. However, for those results the implementation avoided a full search only in case 1a. More detailed experiments are needed with the current implementation on a larger range of sparse examples.

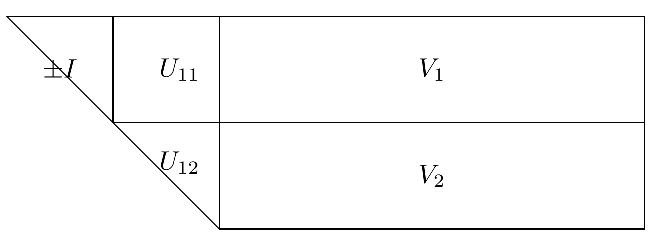
4.2.6 Triangular Preprocessing

As in several other codes, LUSOL's Markowitz strategy has the effect of isolating a "backward triangle" U_1 and a "forward triangle" L_1 before any elimination is performed on the remaining block B. The structure revealed is of the form



With the normal TPP partial pivoting option, LUSOL accepts all rows of U_1 and V as they come, without numerical tests on the pivots. (On the other hand, columns of L_1 and M must satisfy test (4.1) to ensure that L is well-conditioned for LUSOL's updating routines.)

For the TCP option, more care is needed with U_1 to obtain an RRLU factorization. At first sight it seems that all diagonals of U_1 should be subjected to the complete pivoting test (4.3). However, in optimization algorithms it is common for "A" to contain many unit vectors corresponding to slack variables. Ideally they should find their way into U_1, but if A contains a nonzero larger than `FactorTol`, the TCP test would reject them all. Looking more closely, we see that the top rows have the following structure:



The matrix $\pm I$ denotes columns associated with slack variables (and any other unit vectors in A). When the LU factors are used to solve linear systems, each solve with U will involve a system of the form

$$\begin{pmatrix} \pm I & U_{11} & V_1 \\ & U_{12} & V_2 \\ & & U_2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} b \\ c \\ d \end{pmatrix},$$

where z, y and x are determined in that order by back-substitution. As long as the TCP test is applied to all diagonals *after* $\pm I$, any singularities there will be detected and (we assume) corrected. Thus, z and y will not be pathologically large and $x = \pm(U_{11}y + V_1 z)$ will be well defined. (Since the matrices U_{11} and V_1 are original data, they should not contain extremely large numbers.)

4.2.7 The Test for Singularity

After completing a factorization, LUSOL examines the diagonals of U and flags any that are "small" in absolute terms or relative to other elements in the same column of U:

$$\begin{aligned} |U_{jj}| &\leq \mathtt{Utol}_1 \\ \text{or} \quad |U_{jj}| &\leq \mathtt{Utol}_2 \times \max_i |U_{ij}| \end{aligned} \tag{4.4}$$

Typical values for $\mathtt{Utol}_1$ and $\mathtt{Utol}_2$ are $\epsilon^{2/3} \approx 3.7 \times 10^{-11}$ when the factors are being used to solve linear systems. For rank determination in the MATLAB environment, we have used

$$\mathtt{Utol}_1 = n\|A\|\epsilon, \qquad \mathtt{Utol}_2 = 0,$$

with $\|A\|$ obtained from `norm` or `normest` for dense or sparse A respectively. This matches the tolerance MATLAB uses for determining rank via `svd`.

4.3 Rank-Revealing Tests on Classical Examples

For dense matrices, Chan [Cha84], Hwang, Lin, and Yang [HLY92], and Hwang and Lin [HL97] describe two-pass procedures for detecting one or more singularities. These RRLU factorizations reduce P_1AP_2 to $\begin{pmatrix} L & \\ & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix}$, where P_1 and P_2 are permutations and U_{22} is close to zero. The first pass performs an LU factorization using partial pivoting. The second pass uses *inverse iteration* (involving several solves) to find a rank-revealing permutation. The [HLY92] procedure then performs another (modified partial pivoting) LU factorization that preserves this permutation and stops once U_{22} is found (leaving U_{22} unfactorized). The work required is given as $\frac{2}{3}n^3 + 2Jn^2r + 2nr$ operations, where r is the rank-deficiency of the $n \times n$ matrix and J is the number of inverse iterations used.

Since TCP is a variant of LU factorization (producing complete factors L and U), no second pass or inverse iteration is needed to reveal the rank of A. For dense matrices, the work required is also about $\frac{2}{3}n^3$ operations.

Both [Cha84] and [HLY92] give examples of matrices where normal LU factorizations will not detect (near) singularities. In this section, the examples are used to compare five different factorizations:

MD	MATLAB Dense LU
MS	MATLAB Sparse LU
MCS	MATLAB Sparse LU with `colmmd` preprocessing
TPP	Sparse LU with threshold partial pivoting
TCP	Sparse LU with threshold complete pivoting

All MATLAB factorizations use partial pivoting. MS and MCS have a parameter `thresh` that corresponds to 1/`FactorTol`. (See MATLAB help for `lu` and `colmmd` in appendix B.) For these small examples the MATLAB codes used `thresh` $= 1.0$ (the most reliable value), while TPP and TCP used `FactorTol` $= 1.25$ to allow at least a little emphasis on sparsity.

The tests here and later were performed on an SGI Origin 2000 with 195 MHz R10000 CPUs, running Irix 6.5 and MATLAB version 6.0.0.88 (R12) with machine precision $\epsilon = 2^{-52} = 2.2 \times 10^{-16}$.

Example 4.1 [Cha84, p. 543]. The matrix $T_n \equiv T$ with

$$T_{ij} = \begin{cases} 1 & i = j \\ -1 & i < j, \quad 1 \le i, j \le n\ , \\ 0 & i > j \end{cases} \qquad \text{e.g.,} \quad T_4 = \begin{pmatrix} 1 & -1 & -1 & -1 \\ & 1 & -1 & -1 \\ & & 1 & -1 \\ & & & 1 \end{pmatrix},$$

is nearly singular for large n. Chan [Cha84] shows that, for a certain permutation of T_n, an LU factorization has the nth pivot equal to $2^{-(n-2)}$. Once $2^{-(n-2)}$ is less than machine precision, the matrix is effectively singular. Since $\epsilon = 2^{-52}$, T_{50} has effective rank 49. None of the factorizations tested uncovered the singularity correctly. All returned U with a unit diagonal, except MCS, which gave $U_{50,50} = 4.8 \times 10^{-7}$.

This is an extremely pathological case in being already triangular with all nonzero elements of equal magnitude. MD and MS find there is nothing to eliminate, returning $L = I$, $U = T$. MCS happens to reorder the columns and carry out some elimination, but gives only a small hint of singularity.

Some column re-ordering is certainly required (cf. [Cha84]). However, given T with any ordering of its rows and columns, LUSOL reconstructs the original ordering during its "backward triangle" phase. TPP accepts all $+1$ pivots without numerical testing, and TCP would not reject them even with `FactorTol` $= 1.0$.

Example 4.2 [Cha84, p. 544]. This example actually comes from [Wil65, p. 308 and p. 325] and demonstrates that partial pivoting may not produce diagonals small enough to reveal the rank successfully. The 21×21 matrix

$$W = \begin{pmatrix} 10 & 1 & & & & & & & \\ 1 & 9 & 1 & & & & & & \\ & \cdot & \cdot & \cdot & & & & & \\ & & 1 & 1 & 1 & & & & \\ & & & 1 & 0 & 1 & & & \\ & & & & 1 & -1 & 1 & & \\ & & & & & \cdot & \cdot & \cdot & \\ & & & & & & 1 & -9 & 1 \\ & & & & & & & 1 & -10 \end{pmatrix}$$

has one singularity (i.e., exact rank 20). MD and MS both give $|U_{21,21}| = 8.2 \times 10^{-11}$, just missing the indication of singularity. MCS successfully reveals the singularity (one zero pivot), as do TPP and TCP.

Example 4.3 [HLY92, p. 133]. Starting with the matrix $A = \begin{pmatrix} T_{40} & \\ & T_{40} \end{pmatrix}$ and making the following changes:

> **for** $i = 1 : 40,$
> $\quad A(1:40, 81-i) = A(1:40, 81-i) + A(1:40, i)$
> $\quad A(i, 41:80) = A(i, 41:80) + A(81-i, 41:80)$
> **end**

gives a matrix

$$A = \left(\begin{array}{ccccc|ccccc} 1 & -1 & \cdot & \cdot & -1 & -1 & \cdot & \cdot & -1 & 2 \\ & 1 & -1 & & \cdot & \cdot & & \cdot & 2 & -1 \\ & & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ & & & \cdot & -1 & -1 & 2 & \cdot & & \cdot \\ & & & & 1 & 2 & -1 & \cdot & \cdot & -1 \\ \hline & & & & & 1 & -1 & \cdot & \cdot & -1 \\ & & & & & & 1 & -1 & & \cdot \\ & & & & & & & \cdot & \cdot & \cdot \\ & & & & & & & & \cdot & -1 \\ & & & & & & & & & 1 \end{array}\right).$$

The three smallest singular values are $[0.62,\ 1.9 \times 10^{-12},\ 1.9 \times 10^{-12}]$ and Hwang, Lin, and Yang [HLY92] find $U_{22} = \begin{pmatrix} 1.0\times10^{-15} & 3.6\times10^{-12} \\ -3.6\times10^{-12} & 1.3\times10^{-23} \end{pmatrix}$. MD and MS both give $U = A$ (failure). With MCS, the two smallest diagonals of U are $[-0.5,\ 1.5 \times 10^{-11}]$. TPP also produces $U = A$, but TCP gives the last three pivots as $[-0.67,\ -7.3 \times 10^{-12},\ -3.6 \times 10^{-12}]$ (success).

Example 4.4 [HLY92, pp. 133–134]. Starting with the matrix $A = \begin{pmatrix} T_{30} & & \\ & T_{30} & \\ & & T_{30} \end{pmatrix}$ and making the following changes:

```
for i = 1 : 30,
    A(1 : 90, 61 − i) = A(1 : 40, 61 − i) + A(1 : 90, i)
    A(i, 1 : 90) = A(i, 1 : 90) + A(61 − i, 1 : 90)
end
for i = 1 : 30,
    A(1 : 90, 91 − i) = A(1 : 40, 91 − i) + A(1 : 90, 30 + i)
    A(30 + i, 1 : 90) = A(30 + i, 1 : 90) + A(91 − i, 1 : 90)
end
```

gives a matrix

$$A = \left(\begin{array}{ccccc|ccccc|ccccc}
1 & -1 & \cdot & \cdot & -1 & -1 & \cdot & \cdot & -1 & 2 & 2 & -1 & \cdot & \cdot & -1 \\
 & 1 & -1 & & \cdot & \cdot & & \cdot & 2 & -1 & -1 & 2 & \cdot & & \cdot \\
 & & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
 & & & \cdot & -1 & -1 & 2 & \cdot & & \cdot & \cdot & & \cdot & 2 & -1 \\
 & & & & 1 & 2 & -1 & \cdot & \cdot & -1 & -1 & \cdot & \cdot & -1 & 2 \\
\hline
 & & & & & 1 & -1 & \cdot & \cdot & -1 & -1 & \cdot & \cdot & -1 & 2 \\
 & & & & & & 1 & -1 & & \cdot & \cdot & & \cdot & 2 & -1 \\
 & & & & & & & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
 & & & & & & & & \cdot & -1 & -1 & 2 & \cdot & & \cdot \\
 & & & & & & & & & 1 & 2 & -1 & \cdot & \cdot & -1 \\
\hline
 & & & & & & & & & & 1 & -1 & \cdot & \cdot & -1 \\
 & & & & & & & & & & & 1 & -1 & & \cdot \\
 & & & & & & & & & & & & \cdot & \cdot & \cdot \\
 & & & & & & & & & & & & & \cdot & -1 \\
 & & & & & & & & & & & & & & 1
\end{array}\right)$$

with three near-singularities. The four smallest singular values are $[0.40,\ 2.8\times10^{-9},\ 1.4\times10^{-9},\ 1.4\times 10^{-9}]$, and Hwang, Lin, and Yang [HLY92] find U_{22} with values ranging in magnitude from 3.7×10^{-9} to 5.9×10^{-26}. MD and MS both return $U = A$. For MCS the last three pivots (the smallest) were $[0.33,\ 3.3\times10^{-4},\ -1.5\times10^{-5}]$. TPP also produces $U = A$, but TCP gives the last four pivots as $[-0.67,\ -7.4\times10^{-9},\ 3.7\times10^{-9},\ -3.7\times10^{-9}]$.

Example 4.5 [HLY92, pp. 134–136]. Starting with the matrix $A = \begin{pmatrix} W & \\ & W \end{pmatrix}$ and making the following changes:

```
for i = 1 : 21,
    A(1 : 21, 43 − i) = A(1 : 21, 43 − i) + A(1 : 21, i)
    A(i, 22 : 42) = A(i, 22 : 42) + A(43 − i, 22 : 42)
end
```

gives a matrix

$$
A = \left(\begin{array}{ccccccccc|ccccccccc}
10 & 1 & & & & & & & & & & & & & & & 2 & 0 \\
1 & 9 & 1 & & & & & & & & & & & & & 2 & 0 & 2 \\
 & \cdot & \cdot & \cdot & & & & & & & & & & & \cdot & \cdot & \cdot & \\
 & & & 1 & 0 & 1 & & & & & & & 2 & 0 & 2 & & & \\
 & & & & & \cdot & \cdot & \cdot & & & \cdot & \cdot & \cdot & & & & & \\
 & & & & & & 1 & -9 & 1 & 2 & 0 & 2 & & & & & & \\
 & & & & & & & 1 & -10 & 0 & 2 & & & & & & & \\
\hline
 & & & & & & & & & 10 & 1 & & & & & & & \\
 & & & & & & & & & 1 & 9 & 1 & & & & & & \\
 & & & & & & & & & & \cdot & \cdot & \cdot & & & & & \\
 & & & & & & & & & & & & 1 & 0 & 1 & & & \\
 & & & & & & & & & & & & & & \cdot & \cdot & \cdot & \\
 & & & & & & & & & & & & & & & 1 & -9 & 1 \\
 & & & & & & & & & & & & & & & & 1 & -10
\end{array}\right)
$$

with two singularities (i.e., exact rank 40). Hwang, Lin, and Yang [HLY92] find U_{22} with values between 10^{-16} and 10^{-17}. MD and MS both give 8.2×10^{-11} for the last pivot. MCS successfully reveals the singularity (two zero pivots), as do TPP and TCP.

The above examples demonstrate that, of all the factorizations tested, only TCP is likely to be rank-revealing (for all but the most pathological example), as long as `FactorTol` is suitably low. For triangular matrices A or permuted triangles, a necessary condition is $1 \leq \texttt{FactorTol} < \max |A_{ij}|$, which is possible for matrices with at least some variation in the magnitude of the nonzero elements.

4.4 Dynamic Programming Application

The sparse TCP factorization was developed to enhance stability within the computation of Laurent expansion coefficients (see Chapter 3). Dr. Richard Grinold (Barclays Global Investments, San Francisco) provided a large, sparse, substochastic dynamic programming problem, and solving this example by policy improvement requires the Laurent expansion coefficients for many different policies. The method from Chapter 3 finds these coefficients by solving multiple linear systems of varying size, each involving a sparse matrix Q with at most one singularity. Correct rank detection of each Q is crucial for the method. This motivated the sparse, RRLU factorization presented here.

Extracting a subset of the Q matrices from the Barclays example provided a test-bed. Each Q ($n \times n$, $Q \equiv P - I$ for P substochastic) has the following special properties:

- Q has at most 1 singularity;
- $0 \leq q_{ij} \leq 1$, $i \neq j$;
- $q_{ii} = -\sum_{j \neq i} q_{ij}$;

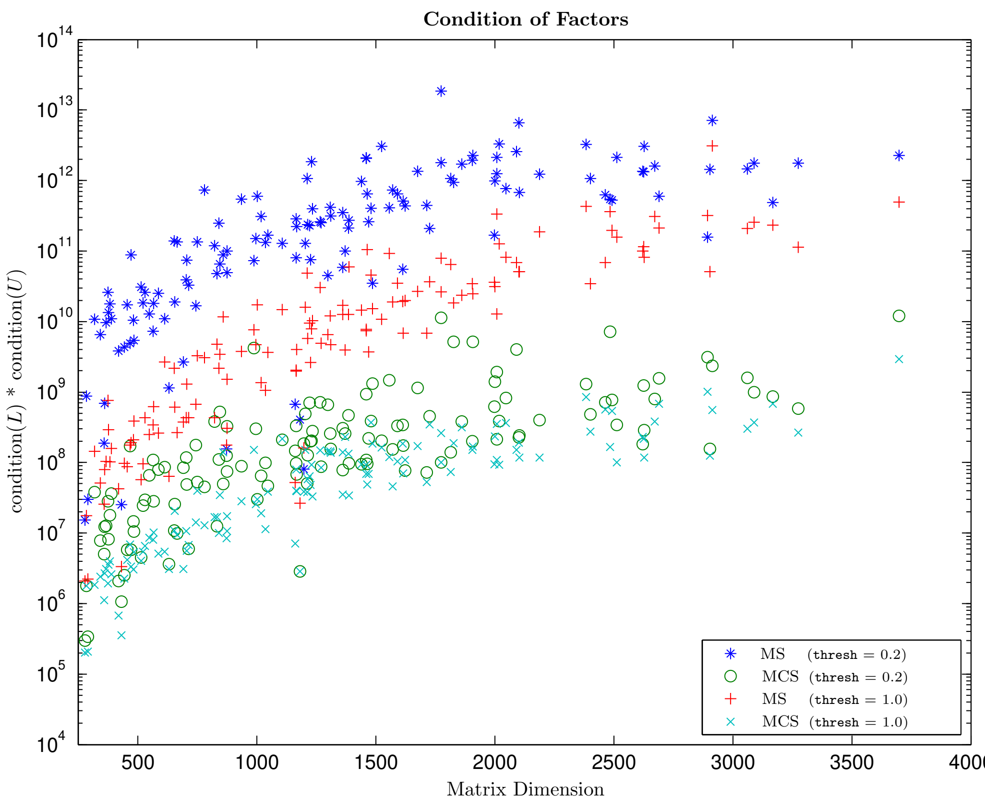


Figure 4.1: Comparing the condition of the factors for MS and MCS

for $1 \leq i, j \leq n$. The same five factorizations used in §4.3 were applied to these matrices, but different values for `thresh`, `FactorTol` and `DropTol` were used. The results confirmed that TCP is at least competitive with all the other factorizations for the measures taken. Comparisons between the factorizations were made using four different measures:

Rank-revealing property Does the diagonal of the U factor reveal the rank of the matrix? As in §4.3, the rank estimation is given by the number of diagonal elements of $U >$ tol $\equiv n \times \|Q\| \times \epsilon$ (see §4.2.7).

Condition of the factors Numerically stable policy improvement (see Chapter 3) solves systems of linear equations using only the the nonsingular submatrices of L and U. If Q is nonsingular, these are the factors L and U in their entirety. If Q has a singularity, then these submatrices are the factors L and U with the last row and column removed. If a factorization is numerically stable, then the condition of the non-singular submatrices should be (relatively) low. The condition of L multiplied by the condition of U gives a single measure of the condition of the

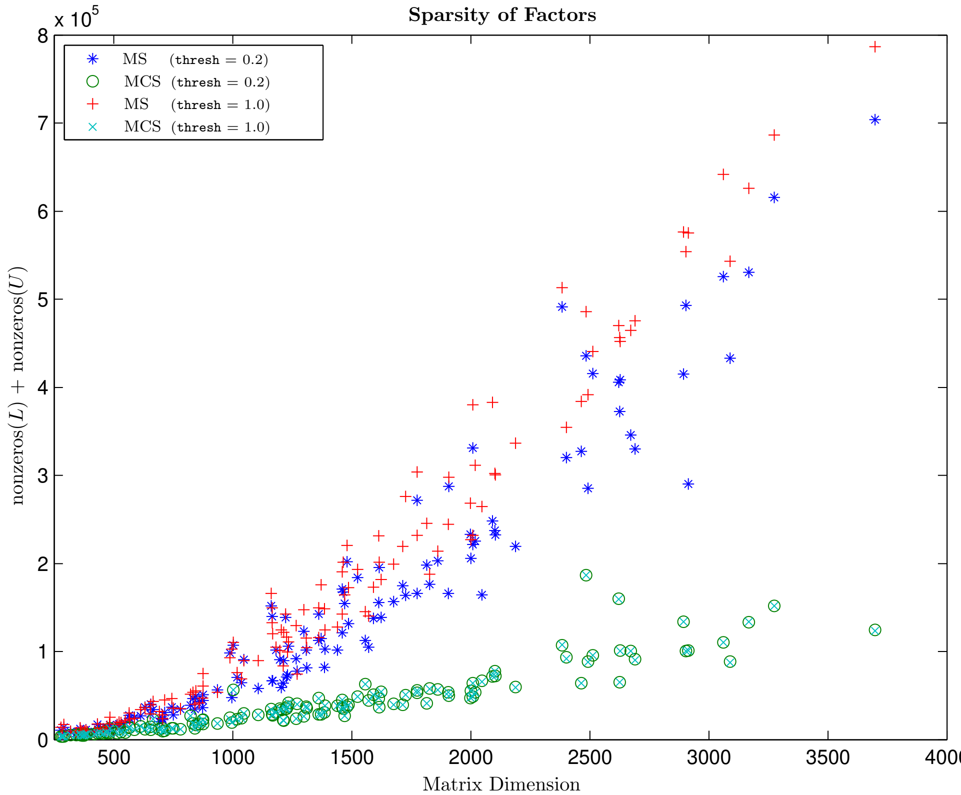


Figure 4.2: Comparing the sparsity of the factors for MS and MCS

factors. Since L and U are sparse, this quantity is estimated by `condest(L) * condest(U)`.[1]

Sparsity of factors Good sparse LU factorizations keep the number of nonzeros in the factors to a minimum. The total number of nonzeros in the factors (i.e., sum of the nonzeros in L and U) gives a measure of the sparsity of the factors.

Time per factorization If a factorization is to be used repeatedly on different matrices (as is the case in many applications of sparse LU factorizations, including the numerically stable policy improvement from Chapter 3), then the time for a single factorization needs to be small. Timing the CPU during each factorization gives this measure.

Figures 4.1-4.3 show a comparison of MS and MCS for two values of `thresh` (0.2 and 1). Notice that for `thresh` = 1, the condition of the factors decreases (representing greater numerical stability). In fact, with `thresh` = 0.2, MS fails to reveal the rank for five of the test matrices. The condition

[1]See MATLAB help for `condest` in appendix B.

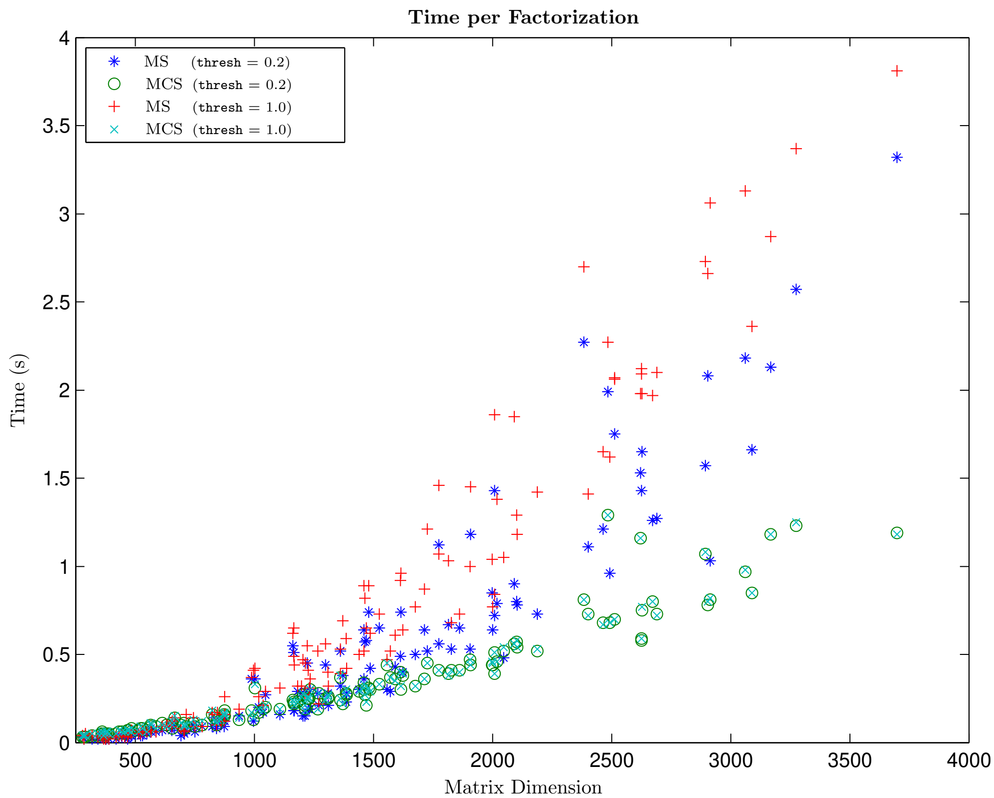


Figure 4.3: Comparing time per factorization for MS and MCS

of the factors for these five cases were so high they are omitted from Figure 4.1 (to provide a clearer comparison of this measure when the factorizations *are* rank-revealing). For both Figure 4.2 and Figure 4.3 different values of `thresh` give the same result for MCS. However, different values of `thresh` affect MS in the expected way (i.e., higher values for `thresh` improve the condition of the factors, decrease the sparsity of the factors, and slow the factorization down). All three figures show that MCS works considerably better as a sparse rank-revealing factorization for the given test matrices. Also, it shows that using `thresh` $= 1$ improves the conditions of the factors without significantly decreasing sparsity or slowing the factorization down. Hereafter, MCS refers to MATLAB Sparse LU with `colmmd` preprocessing and `thresh` $= 1$.

Of the other factorizations tested, TPP and TCP were tested using `FactorTol` $= 5$ and 10. These factorizations along with MD all successfully revealed the rank of every test matrix. It may be that the special structure of the test matrices allows partial pivoting to become rank revealing.

Figure 4.4 shows that TPP performs the worst numerically (for `FactorTol` $= 5$ and 10), but TCP with `FactorTol` $= 5$ and MCS both perform almost as well as MD using this measure. For the

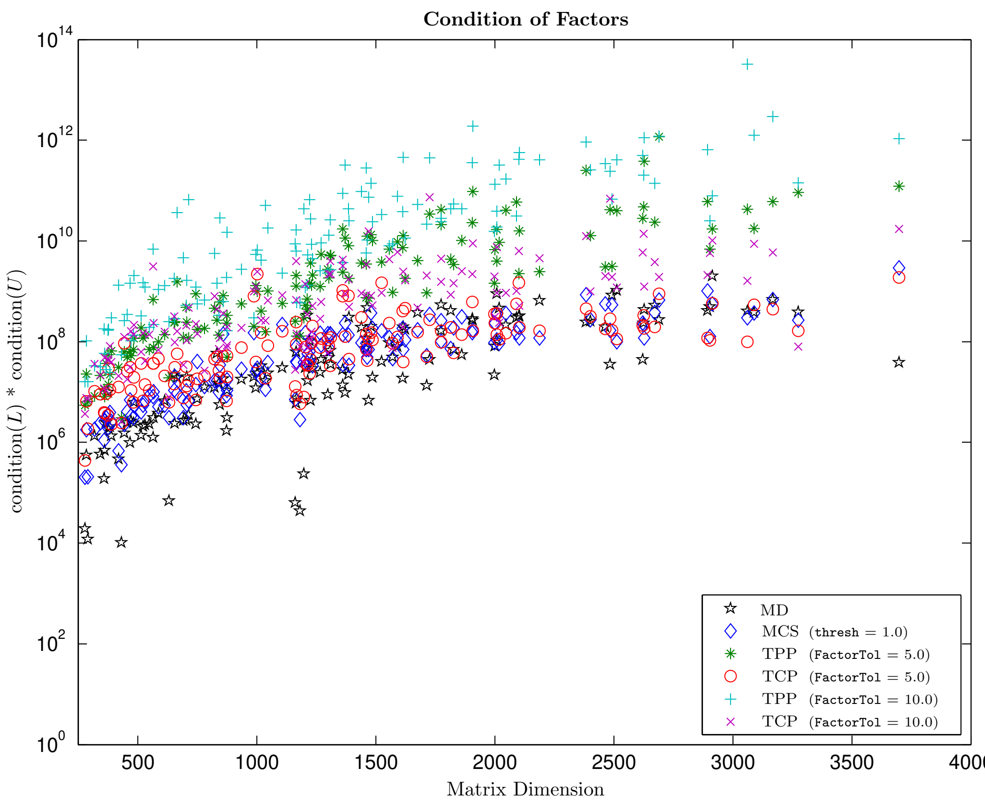


Figure 4.4: Comparing the condition of the factors

sparsity of factors and the time per factorization measures, MD is omitted since it performs so poorly as to obscure differences between the other factorizations. When comparing the factorizations using the sparsity of factors measure (see Figure 4.5), the performance order is reversed. TPP performs the best (with either `FactorTol`), then TCP, and lastly MCS. Even with `FactorTol` = 5, TCP shows a considerable improvement over MCS.

Next, Figure 4.6 compares the time per factorization and shows the one weakness of TCP. Even with `FactorTol` = 10 (allowing more flexibility in choosing diagonals, and therefore faster execution), the time per factorization appears to be growing faster for TCP than either MCS or TPP. It appears that both TCP and MCS take polynomial time with respect to the number of nonzeros in the original matrix, whereas TPP appears to grow linearly.

A further test helped discern the merits of the different sparse LU factorizations. The largest test matrix was used (3699×3699 with one singularity). With `FactorTol` ranging from 1 to 50 (and `thresh` = 1/`FactorTol`), it became clear how MCS, TPP and TCP could be changed to improve performance in any of the measures used. All successfully reveal the rank of the matrix, but it

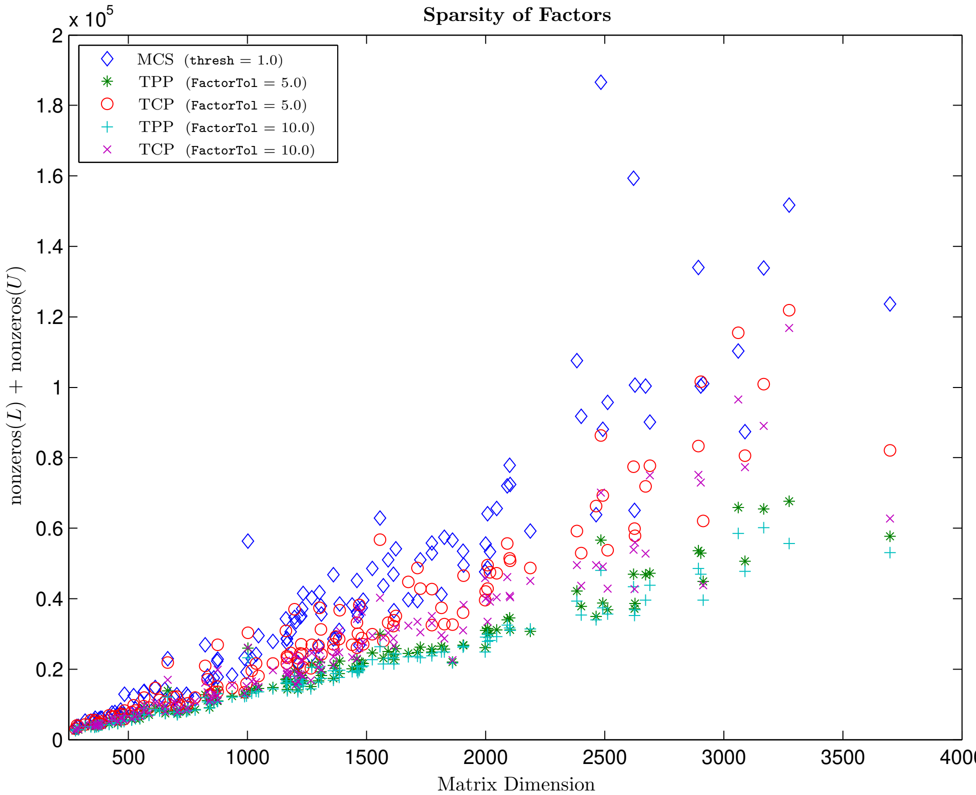


Figure 4.5: Comparing the sparsity of the factors

appears that finding the rank of these particular test matrices is not difficult. Figures 4.7–4.9 shows that for MCS only the condition of the factors is affected by changes in `thresh`. Setting `thresh` = 1 appears to give the optimal performance for MCS. For both TPP and TCP, the condition of the factors improves as `FactorTol` drops towards 1, but the sparsity of the factors and time per factorization both deteriorate. It appears that `FactorTol` = 10.0 would be a reliable yet efficient value.

Finally, Figure 4.10 compares the running times of MCS, TPP, and TCP. It also shows $O(p)$ and $O(p^2)$ times (the lower line and upper line, respectively), where $p = \mathrm{nnz}(A)$, the number of nonzeros in A. The running time for TPP appears to be $O(p)$ (very satisfactory). TCP appears to be competitive with MCS (with more variability), but both factorizations display times that are $O(p^k)$, $1 < k < 2$. The cost of maintaining $A_{\max}$ is evident. On the other hand, since $p \ll m \times n$ if catastrophic fill-in doesn't occur, these factorizations will generally perform better than the $O(n^3)$ time for dense LU factorizations.

Figure 4.11 changes the horizontal axis of Figure 4.10 from the number of nozeros in A to the

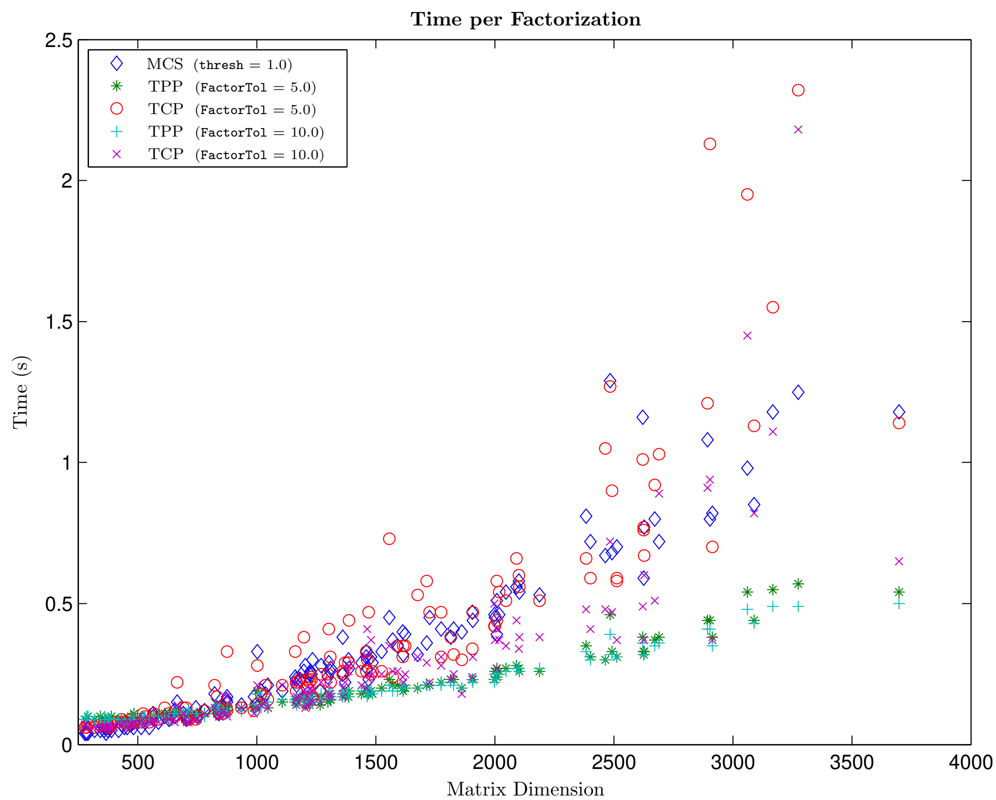


Figure 4.6: Comparing the time per factorization

sparsity of the factors (total nonzeros in L and U). Only the TCP factorization (with `FactorTol` = 5.0 and 10.0) are shown, along with $O(p)$ and $O(p^2)$ times. Now $p = \text{nnz}(L+U)$ represents density of the factors. Ideally the running time should be $O(p)$, but it appears that TCP is again $O(p^k)$ for $1 < k < 2$. TCP also runs in polynomial time with respect to the sparsity of its factors, but the degree of the polynomial is less than 2.

4.5 Optimization Application

One major application of sparse LU factorizations is within large-scale optimization systems. In particular, LUSOL is used within MINOS and SNOPT [MS83, GMS97] to implement the simplex method and various quadratic programming and nonlinear programming algorithms. In this context, if a basis matrix appears to be singular, certain columns are replaced by unit vectors (corresponding to slack variables) and the factorization is repeated. LUSOL signals singularity or ill-conditioning by flagging small diagonals of U as described in §4.2.7. The new TCP option has proved invaluable

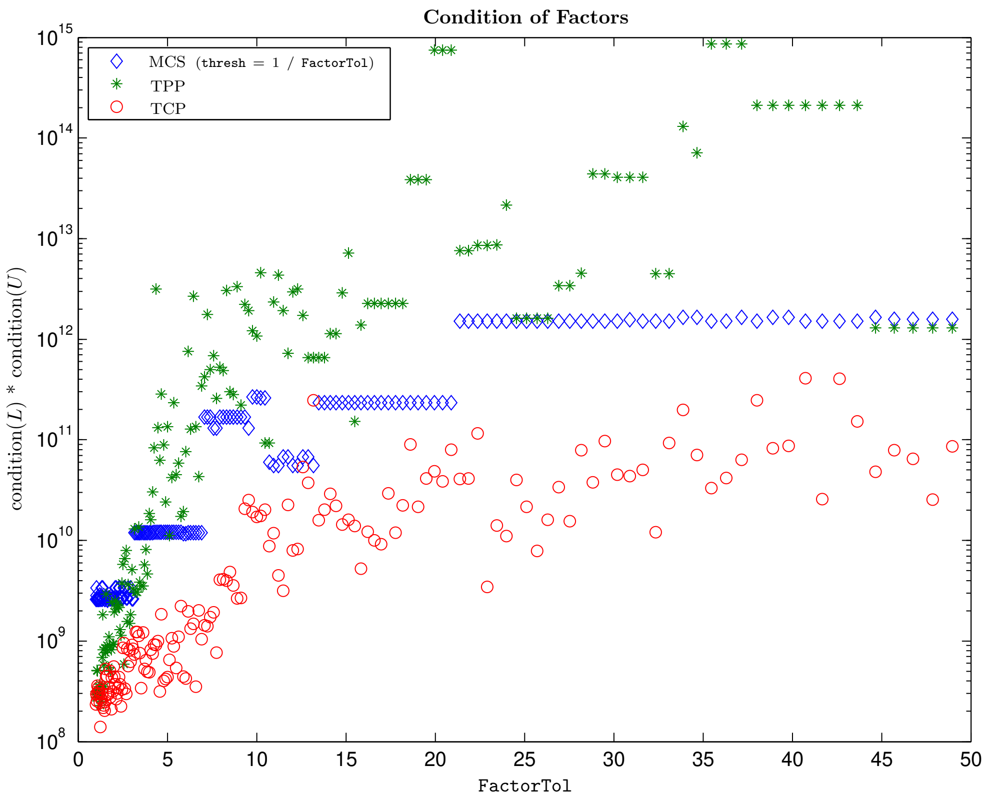


Figure 4.7: Comparing the condition of the factors as `FactorTol` changes

for ensuring reliability in these tests. Preliminary experience with SNOPT is described in [GMS02].

In particular, the first basis matrix is often ill-conditioned as it is chosen heuristically by the optimizer's Crash procedure, or input by the user (with unpredictable properties). In one case with $n = 10000$, the usual TPP factorization with `FactorTol`$= 4.0$ indicated 243 singularities. After slacks were inserted, the next factorization indicated 47 additional singularities, the next a further 25, then 18, 14, 10, and so on. Nearly 30 TPP factorizations and 460 new slacks were required before the basis was regarded as suitably nonsingular. Since L and U each had about a million nonzeros in all factorizations, the repeated failures were rather expensive. In contrast, a single TCP factorization with `FactorTol` $= 2.5$ indicated 100 singularities, after which the modified B proved to be very well conditioned. Although L and U were more dense (1.35 million nonzeros each) and much more expensive to compute, the subsequent optimization required significantly fewer major and minor iterations.

In general, TCP is the primary recourse when unexpected growth occurs in $\|x\|$ following solution of $Ax = b$. (If necessary, `FactorTol` is gradually reduced towards 1.1 until x appears to be accurate.)

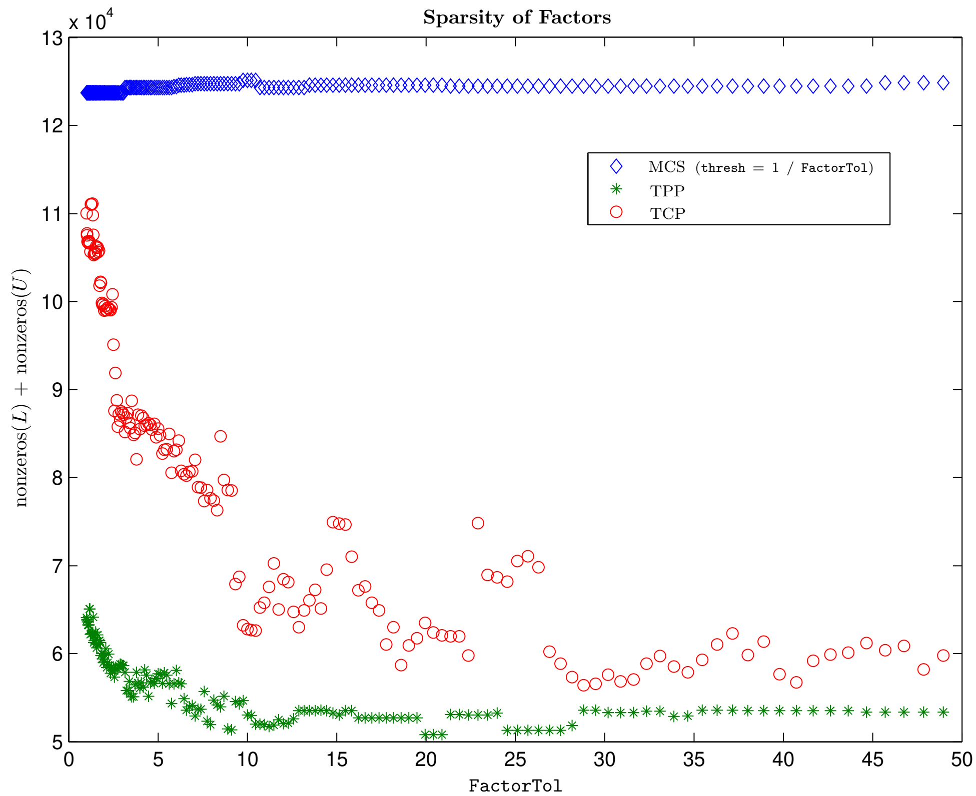


Figure 4.8: Comparing the sparsity of the factors as `FactorTol` changes

The new option has proved valuable for some optimization problems arising from partial differential equations. A regular "marching pattern" is sometimes present in A, particularly in the first basis from Crash (as mentioned). With threshold partial pivoting the factors display no small diagonals in U, yet the TCP factors reveal a large number of dependent columns.

4.6 Contributions

This chapter presents a new pivoting strategy for sparse LU factorizations, *threshold complete pivoting* (TCP). As in the dense case, this pivoting strategy is likely to be more stable than the traditional threshold partial pivoting, and most importantly it appears to be rank-revealing for all but the most pathological matrices. The rank-revealing property is discussed and compared for some classical examples as well as the DP application that motivated developing TCP. Finally, this chapter describes the benefits of TCP within classical optimization algorithms.

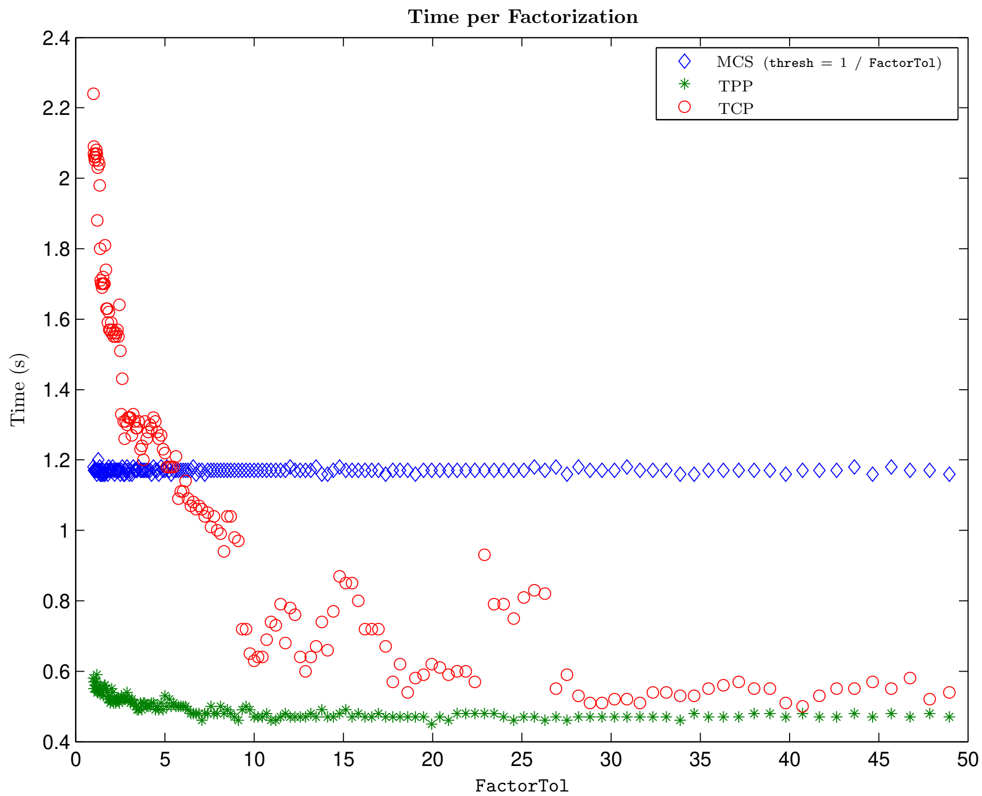


Figure 4.9: Comparing the time per factorization as `FactorTol` changes

Acknowledgement The work contained within this chapter is joint work with Prof. Michael A. Saunders.

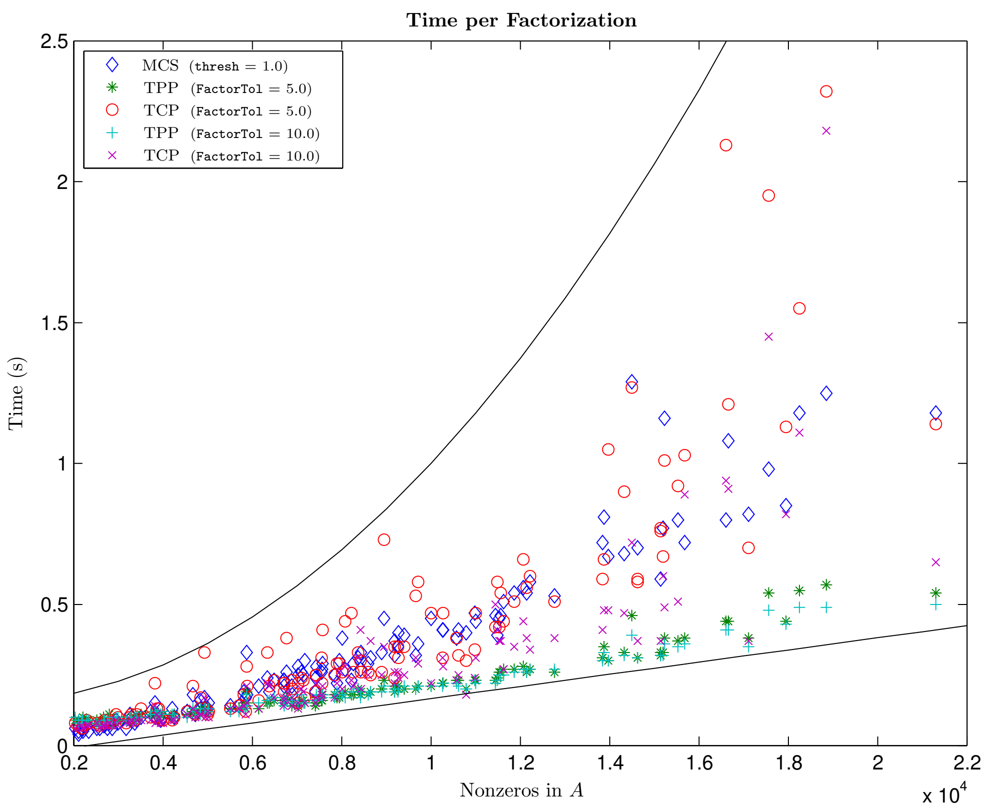


Figure 4.10: Time per factorization as a function of the nonzeros in A

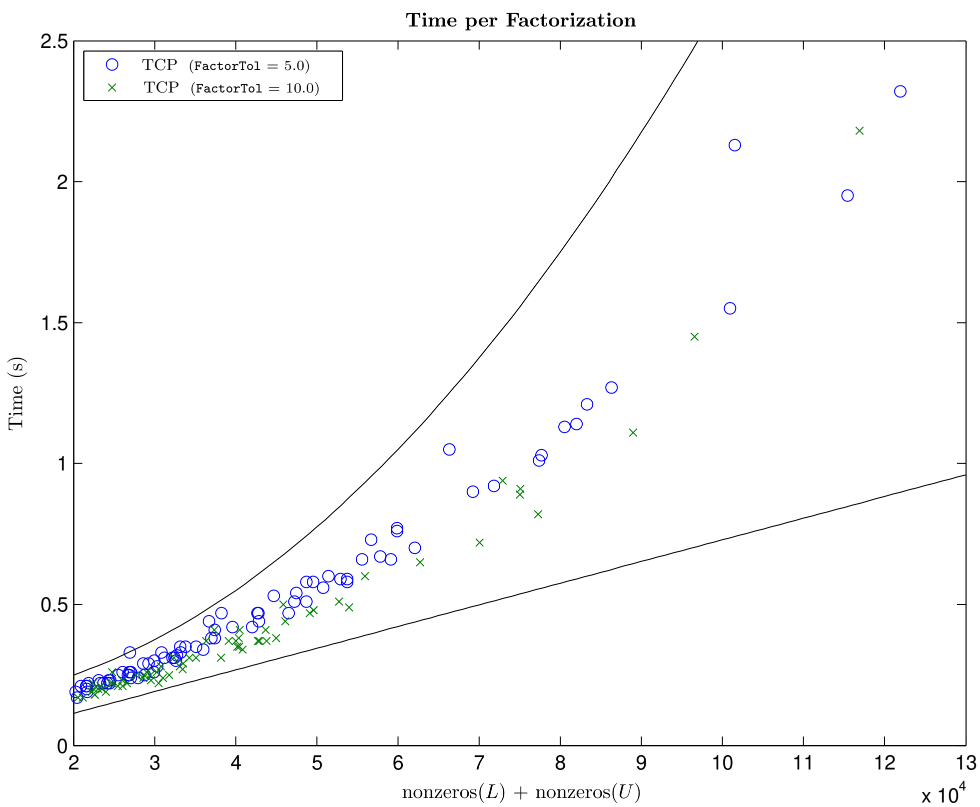


Figure 4.11: Time per factorization as a function of the sparsity of the factors

Appendix A

Counterexample to Method for Finding 1-Optimal Policies

Denardo [Den71, p. 491] sketches two methods he states will find a 1-optimal policy. One method uses policy improvement and the other uses linear programming together with some auxiliary routines. Both methods can fail to find a 1-optimal policy as the following example shows.

Example A.1 Consider the deterministic system in Figure A.1. A policy is an ordered pair of

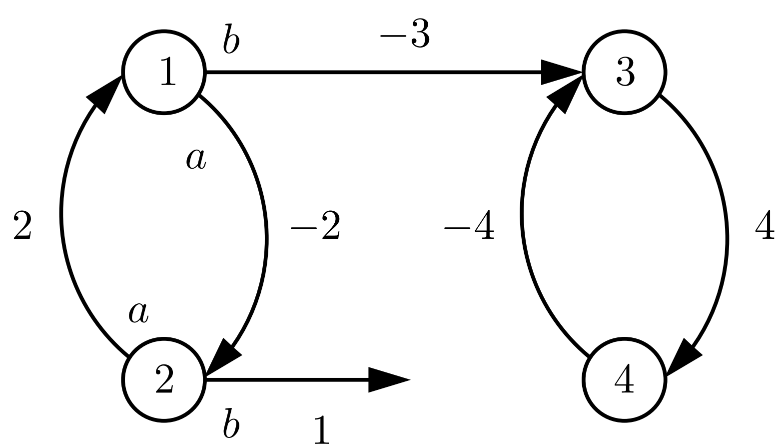


Figure A.1: Counterexample for Denardo's Method

actions for the states 1 and 2. For example, ba is the policy that takes action b in state 1 and action a in state 2. In this example, all policies are 0-optimal, but only $\alpha \equiv aa$ is 1-optimal.

Starting with $\delta \equiv bb$ and implementing either of Denardo's methods shows first that δ is 0-optimal [Den70a]. Next Denardo finds a column S-vector v^1 that satisfies $Q_\delta v^1 = v^0_\delta$. One such vector is $v^1 = \begin{pmatrix} 1 & -1 & 0 & 2 \end{pmatrix}^T$. He then defines the set E^1 of policies γ for which $Qv^1 = v^0_\delta$ and solves a maximum-reward-rate problem over E^1. Since E^1 is the singleton set $\{\delta\}$, δ is the unique solution of that problem whether solved by policy improvement or linear programming. The final step of each of Denardo's methods finds a policy γ that is transient on states where δ is transient, viz., states 1 and 2. Consequently, γ cannot be α because states 1 and 2 are recurrent under α. Thus Denardo's methods both fail to find a 1-optimal policy.

Appendix B

MATLAB Help Files

```
LU LU factorization.

     [L,U] = LU(X) stores an upper triangular matrix in U and a
     "psychologically lower triangular matrix" (i.e.  a product of
     lower triangular and permutation matrices) in L, so that X =
     L*U. X must be square.

     [L,U,P] = LU(X) returns lower triangular matrix L, upper
     triangular matrix U, and permutation matrix P so that P*X = L*U.

     LU(X), with one output argument, returns the output from LAPACK'S
     DGETRF or ZGETRF routine.

     LU(X,THRESH) controls pivoting in sparse matrices, where THRESH
     is a pivot threshold in [0,1].  Pivoting occurs when the diagonal
     entry in a column has magnitude less than THRESH times the
     magnitude of any sub-diagonal entry in that column.
     THRESH = 0 forces diagonal pivoting.  THRESH = 1 is the default.

     See also LUINC, QR, RREF.
```

```
COLMMD Column minimum degree permutation.

     P = COLMMD(S) returns the column minimum degree permutation
     vector for the sparse matrix S. For a non-symmetric matrix S,
     S(:,P) tends to have sparser LU factors than S.

     See also SYMMMD, SYMRCM, COLPERM.
```

```
CONDEST 1-norm condition number estimate.

    C = CONDEST(A) computes a lower bound C for the 1-norm
    condition number of a square matrix A.

    C = CONDEST(A,T) changes T, a positive integer parameter equal
    to the number of columns in an underlying iteration matrix.
    Increasing the number of columns usually gives a better condition
    estimate but increases the cost.  The default is T = 2, which
    almost always gives an estimate correct to within a factor 2.

    [C,V] = CONDEST(A) also computes a vector V which is an
    approximate null vector if C is large.  V satisfies NORM(A*V,1)
    = NORM(A,1)*NORM(V,1)/C.

    Note:  CONDEST invokes RAND. If repeatable results are required
    then invoke RAND('STATE',J), for some J, before calling this
    function.

    Uses block 1-norm power method of Higham and Tisseur.

    See also NORMEST1, COND, NORM.
```

Bibliography

[AA98] K. E. Avrachenkov and E. Altman, *Sensitive discount optimality via nested linear programs for ergodic Markov decision processes*, Unpublished manuscript, 1998.

[AS80] B. Aspvall and Y. Shiloach, *A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality*, SIAM J. Comput. **9** (1980), no. 4, 827–845.

[Bal61] M. Balinski, *On Solving Discrete Stochastic Decision Problems*, Study 2. Mathematica, Princeton, New Jersey, 1961.

[Bat73] J. A. Bather, *Optimal decision procedures for finite Markov chains. Part III: General convex systems*, Adv. Appl. Prob. **5** (1973), 541–558.

[Bel58] R. A. Bellman, *On a routing problem*, Quart. Appl. Math. **16** (1958), 87–90.

[Bla62] D. Blackwell, *Discrete dynamic programming*, Ann. Math. Statist. **33** (1962), no. 2, 719–726.

[Bus71] P. A. Businger, *Monitoring the numerical stability of Gaussian elimination*, Numer. Math. **16** (1971), 360–361.

[Cha84] T. F. Chan, *On the existence and computation of LU-factorizations with small pivots*, Math. Comp. **42** (1984), no. 166, 535–547.

[d'E60] F. d'Epenoux, *Sur un problème de production et de stockage dans l'aléatoire*, Revue Française de Recherche Opérationnelle **4** (1960), no. 14, 3–13, An English translation appears as: A Probabilistic Production and Inventory Problem. Management Sci. 10, 1, 98–108.

[DEG+99] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, Xiaoye S. Li, and J. W. H. Liu, *A supernodal approach to sparse partial pivoting*, SIAM J. Matrix Anal. Appl. **20** (1999), no. 3, 720–755.

[Den70a] E. V. Denardo, *Computing a bias-optimal policy in a discrete-time Markov decision problem*, Oper. Res. **18** (1970), 279–289.

[Den70b] ———, *On linear programming in a Markov decision problem*, Management Sci. **16** (1970), no. 5, 281–288.

[Den71] ———, *Markov renewal programs with small interest rates*, Ann. Math. Statist. **42** (1971), no. 2, 477–496.

[Der62] C. Derman, *On sequential decisions and Markov chains*, Management Sci. **9** (1962), no. 1, 16–24.

[DF68] E. V. Denardo and B. L. Fox, *Multichain Markov renewal programs*, SIAM J. Appl. Math. **16** (1968), no. 3, 468–487.

[dG60] G. de Ghellinck, *Les problèms de décisions séquentielles*, Cahiers du Centre d'Etudes de Recherche Opérationnelle **2** (1960), no. 2, 161–179.

[DR96] I. S. Duff and J. K. Reid, *The design of MA48: A code for the direct solution of sparse unsymmetric linear systems of equations*, ACM Trans. Math. Software **22** (1996), no. 2, 187–226.

[Duf77] I. S. Duff, *MA28—a set of Fortran subroutines for sparse unsymmetric linear equations*, Report AERE R8730, Atomic Energy Research Establishment, Harwell, England, 1977.

[EV75] B. C. Eaves and A. F. Veinott, Jr., *Policy improvement in positive, negative and stopping Markov decision chains*, Unpublished manuscript, 1975.

[For56] L. R. Ford, *Network flow theory*, Tech. Report P-923, The RAND Corporation, Santa Monica, CA, 1956.

[Fos97] L. V. Foster, *The growth factor and efficiency of Gaussian elimination with rook pivoting*, J. Comput. Appl. Math. **86** (1997), 177–194.

[GMS97] P. E. Gill, W. Murray, and M. A. Saunders, *User's guide for SNOPT 5.3: A Fortran package for large-scale nonlinear programming*, Numerical Analysis Report 97-5, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997, Revised May 1998.

[GMS02] ———, *SNOPT: An SQP algorithm for large-scale constrained optimization*, SIAM J. Optim. (2002).

[GMSW87] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright, *Maintaining LU factors of a general sparse matrix*, Linear Algebra Appl. **88/89** (1987), 239–270.

[Gup00] A. Gupta, *WSMP: Watson Sparse Matrix Package, Part II—direct solution of general sparse systems*, IBM Research Report RC 21888 (98472), T. J. Watson Research Center, Yorktown Heights, NY, 2000, `http://www.cs.umn.edu/~agupta/wsmp.html`.

[HA99] M. Hartman and C. Arguelles, *Transience bounds for long walks*, Math. Oper. Res. **24** (1999), 414–439.

[HK79] A. Hordijk and L. C. M. Kallenberg, *Linear programming and Markov decision chains*, Management Sci. **25** (1979), no. 4, 352–362.

[HL97] T. Hwang and W. Lin, *Improved bound for rank-revealing LU factorizations*, Linear Algebra Appl. **261** (1997), 173–186.

[HLY92] T. Hwang, W. Lin, and E. K. Yang, *Rank-revealing LU factorizations*, Linear Algebra Appl. **175** (1992), 115–141.

[HN94] D. S. Hochbaum and J. Naor, *Simple and fast algorithms for linear and integer programs with two variables per inequality*, SIAM J. Comput. **23** (1994), no. 6, 1179–1192.

[How60] R. A. Howard, *Dynamic Programming and Markov Processes*, Technology Press-Wiley, Cambridge, Massachusetts, 1960.

[HP91] M. Haviv and M. L. Puterman, *An improved algorithm for solving communicating average reward Markov decision processes*, Ann. Oper. Res. **28** (1991), 229–242.

[HT00] N. J. Higham and F. Tisseur, *A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra*, SIAM J. Matrix Anal. Appl. **21** (2000), no. 4, 1185–1201.

[Kar78] R. M. Karp, *A characterization of the minimum cycle mean in a digraph*, Discrete Math. **23** (1978), 309–311.

[Kha79] L. G. Khachian, *A polynomial algorithm for linear programming*, Doklady Akad. Nauk SSSR **244** (1979), no. 5, 1093–1096, English translation in Soviet Math. Doklady 20, 191–194.

[Man60] A. S. Manne, *Linear programming and sequential decisions*, Management Sci. **6** (1960), no. 3, 259–267.

[Mar57] H. M. Markowitz, *The elimination form of inverse and its applications to linear programming*, Management Sci. **3** (1957), 255–269.

[MS83] B. A. Murtagh and M. A. Saunders, *MINOS 5.0 User's Guide*, Report SOL 83-20, Department of Operations Research, Stanford University, Stanford, CA, 1983.

[MV69] B. L. Miller and A. F. Veinott, Jr., *Discrete dynamic programming with a small interest rate*, Ann. Math. Statist. **40** (1969), no. 2, 366–370.

[Rei82] J. K. Reid, *A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases*, Math. Prog. **24** (1982), 55–69.

[Rot75] U. G. Rothblum, *Normalized Markov decision chains I: Sensitive discount optimality*, Oper. Res. **23** (1975), no. 4, 785–795.

[RV91] K. W. Ross and R. Varadarajan, *Multichain Markov decision processes with a sample path constraint: A decomposition approach*, Math. Oper. Res. **16** (1991), no. 1, 195–207.

[SS90] U. H. Suhl and L. M. Suhl, *Computing sparse LU factorizations for large-scale linear programming bases*, ORSA J. Comput. **2** (1990), 325–335.

[Tar72] R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput. **1** (1972), 146–160.

[Tre87] W. F. Trench, *A note on solving nearly triangular Toeplitz systems*, Linear Algebra Appl. **93** (1987), 57–65.

[Vei66] A. F. Veinott, Jr., *On finding optimal policies in discrete dynamic programming with no discounting*, Ann. Math. Statist. **37** (1966), no. 5, 1284–1294.

[Vei68] ______, *Discrete dynamic programming with sensitive optimality criteria*, Ann. Math. Statist. **39** (1968), 1372, Preliminary Report.

[Vei69] ______, *Discrete dynamic programming with sensitive discount optimality criteria*, Ann. Math. Statist. **40** (1969), no. 5, 1635–1660.

[Vei74] ______, *Markov decision chains*, Studies in Optimization. MAA Studies in Mathematics (B. C. Eaves and G. B. Dantzig, eds.), vol. 10, Mathematical Association of America, 1974, pp. 124–159.

[Wil65] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965.

[Zla80] Z. Zlatev, *On some pivotal strategies in Gaussian elimination by sparse technique*, SIAM J. Numer. Anal. **17** (1980), no. 4, 18–30.

[ZWS81] Z. Zlatev, J. Wasniewski, and K. Schaumburg, *Y12M: Solution of Large and Sparse Systems of Linear Algebraic Equations*, Number 121 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1981.