AgEcon SEARCH
RESEARCH IN AGRICULTURAL & APPLIED ECONOMICS

The World's Largest Open Access Agricultural & Applied Economics Digital Library

This document is discoverable and free to researchers across the globe due to the work of AgEcon Search.

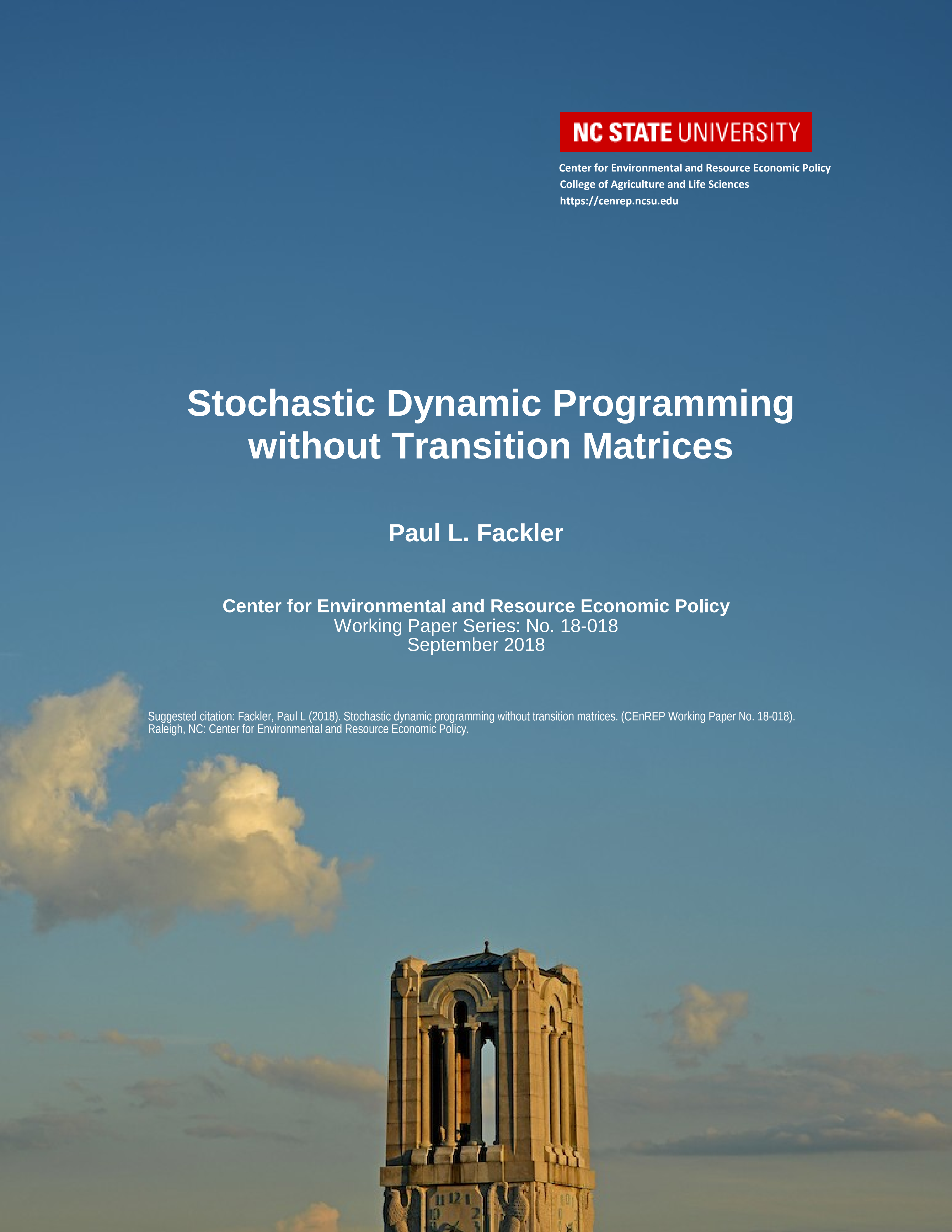
Help ensure our sustainability.

Give to AgEcon Search

AgEcon Search
http://ageconsearch.umn.edu
aesearch@umn.edu

*Papers downloaded from **AgEcon Search** may be used for non-commercial purposes and personal study only. No other use, including posting to another Internet site, is permitted without permission from the copyright owner (not AgEcon Search), or as allowed under the provisions of Fair Use, U.S. Copyright Act, Title 17 U.S.C.*

No endorsement of AgEcon Search or its fundraising activities by the author(s) of the following work or their employer(s) is intended or implied.



NC STATE UNIVERSITY

Center for Environmental and Resource Economic Policy
College of Agriculture and Life Sciences
https://cenrep.ncsu.edu

Stochastic Dynamic Programming without Transition Matrices

Paul L. Fackler

Center for Environmental and Resource Economic Policy
Working Paper Series: No. 18-018
September 2018

Suggested citation: Fackler, Paul L (2018). Stochastic dynamic programming without transition matrices. (CEnREP Working Paper No. 18-018). Raleigh, NC: Center for Environmental and Resource Economic Policy.

Stochastic dynamic programming without transition matrices

Paul L. Fackler
Professor, North Carolina State University
9/17/2018

Abstract: Discrete dynamic programming, widely used in addressing optimization over time, suffers from the so-called curse of dimensionality, the exponential increase in problem size as the number of system variables increases. One method to partially address this problem is to avoid the use of state transition probability matrices, which grow in the square of the size of the state space. This can be done through the use of expected value (EV) functions, which compute the expectation of functions of the future state variables conditioned on current variables. Two ways that this leads to potential gains arise when the state transition can be broken into separate phases and when the transitions for different state variables are conditionally independent. Both of these situations arise in models that are used in natural resource management and are illustrated with several examples.

Keywords: dynamic programming, computational efficiency, SPOMs, harvest management, Kronecker products

Discrete dynamic programming (DDP) is a fundamental tool for making good decisions concerning dynamically changing systems. For a gentle introduction see Marescot, et al. (2013) and for more in-depth discussions see Puterman (1994) or Rust (2008). A significant limitation of DDP, the so-called curse of dimensionality, arises due to the exponential increase in the problem size as the number of variables increases (Powell and Topaloglu; 2005). This problem is particularly acute in the handling of the state transition, which is typically defined in terms of a transition probability matrix P that specifies the probability that some specific value of the state variable will occur in the next period given the current value of the state and actions variables. The total number of elements of this matrix grows with the square of the number of state values.

This note discusses how the curse of dimensionality can be made somewhat less problematic by careful attention to how the transition is handled. In particular it points out that the transition matrix P need not be explicitly defined but instead can be replaced by a function which computes the expectation of future values conditioned on current states and actions. Such a function will be referred to as an

© 2017-2018, Paul L. Fackler

expected value (EV) function and its use can have significant advantages both in reducing memory requirements and in speeding up computations (using both function and policy iteration). Two common examples of when such advantages are possible arise when the state variables are conditionally independent or when the transition can be broken into separate phases. The methods discussed in this note are easily implemented using the freely available MATLAB based MDPSolve package (Fackler, 2011) and code for the examples discussed here is available as a supplement.

One application area where the curse of dimensionality is particularly problematic is in conservation management of spatial units. For example in Stochastic Patch Occupancy Models (SPOMs) the state variables are binary variables representing the absence or presence of some species on a site (patch). With N sites the state space is 2^N and thus grows exponentially in the number of sites. Another example of such a problem is the reserve site selection problem in which a set of sites are targeted for acquisition by a conservation organization but may instead be acquired and developed for non-conservation uses. In this case each site has three alternative states (available, reserved, developed) and hence the state space is 3^N.

This paper first briefly reviews the dynamic programming framework, including a discussion of how index vectors can be used to improve efficiency. It then introduces the concept of an Expected Value (EV) function. Two situations which lead to significant advantages by using EV functions are then discussed and illustrated. The first is the situation in which the state transition occurs in stages, with each stage represented by a sparse transition probability matrix. The second is when a model can be represented in a factored form by a set of conditionally independent state transitions.

Dynamic Programming

The basic components of a DDP model are (1) a reward function $R(S, A)$ which describes the current net benefits of being in a given state S and taking a specified action A, (2) a transition probability matrix, $P(S^+|S, A)$, which gives the transition probability of moving to a specified state S^+ in the next period, given the current state and action and (3) a discount factor $\delta \in [0,1]$ that measures the value of obtaining a given reward in the next period relative to obtaining it this period. The solution to a dynamic programming problem is a strategy that defines how the action should be chosen for each value of the state, $A^*(S)$, and a value function $V(S)$ which describes the value in each state of the sum of the expected discounted rewards when using the optimal strategy.

Standard algorithms for solving dynamic programming problems are based on the Bellman Equation

$$V(S) = \max_A R(S,A) + \delta \sum_{S^+} P(S^+|S,A)V(S^+)$$

If there are n_s values of the state variable(s) and n_x possible combinations of state and action values then V is an n_s element vector, R is an n_x element vector and P is an $n_s \times n_x$ column-stochastic matrix (a matrix composed of non-negative numbers with columns that sum to 1).[2] In this case the Bellman function can be written as

$$V = \max_A R_A + \delta P_A^\top V^+$$

where the A refers to a given strategy. The two standard methods for solving DP problems (function and policy iteration) both use an initial guess of the vector V and compute the vector $\tilde{V} = R + \delta P^\top V$. Each row of this vector is associated with a specified value for the state and the maximal value for each state can then be identified. This results in an n_s vector of indices I^a that selects these values of $\tilde{V}$

$$I_i^a = \underset{j:\, I_x(j)=i}{\operatorname{argmax}} \tilde{V}_j$$

(the use of index vectors is discussed in more detail in Supplemental Appendix 1). The two methods differ in how they update V. Function iteration replaces V with $\tilde{V}[I^a]$ whereas policy iteration replaces V with the solution to the linear system

$$(I - \delta P[:, I^a]^\top)V = R[I^a].$$

[2] Alternatively it could be an $n_x \times n_s$ row-stochastic matrix with rows that sum to 1.

Both methods repeat this process iteratively until a convergence criterion is met. In general, policy iteration uses fewer iterations but each iteration is more expensive because of the need to perform a linear solve.

The state S is typically composed of a set of d_s variables and the size of the state space is the number of possible combinations (tuples) of these variables. A significant challenge in formulating and solving realistic decision models is the so-called curse of dimensionality. The problem size grows exponentially as d_s increases; for example, if all state variables can take on m different values then the size of the state space is m^{d_s}. Of particular importance is that the P matrix can become prohibitively large. Even when sparse (i.e., having many 0 elements) it can use up large amounts of memory and performing the linear solve in policy iteration may become extremely time consuming or even impossible due to memory limitations. Even the matrix-vector operations used to compute $P^{\top}V$ may be prohibitively time-consuming.

One approach to rescuing policy iteration which works well for large problems uses iterative linear solvers, including Krylov methods (Barrett et al., 1994). This approach is discussed in Rust (1996) and was demonstrated by Mrkaic (2002) to result in significant reductions in the time required for each iteration when using policy iteration. The use of Krylov methods, such as Generalized Minimum Residual (GMRES) and Bi-Conjugate Gradient-Stabilized (BiCGSTAB), are easily implemented into dynamic programming algorithms in MATLAB because these linear equation solvers are part of the basic MATLAB package.

What does not appear to be widely recognized in the literature is the potential for memory and speed efficiencies from not forming the P matrix in the first place. All that is required of function iteration or policy iteration, if a Krylov solver is used, is that $P^{\top}V$ can be evaluated.

Expected Value Functions

An expected value (EV) function produces the same result as $P^\top V$ but without the need to explicitly compute P. Specifically, an EV function v transforms the future state vector into its expectation conditional on current states and actions (X):

$$v(V^+) = E[V^+|X]$$

An EV function might also use a second input argument,

$$v(V^+, I^a) = E[V^+|X[I^a, :]]$$

in which case it is an indexed evaluation that transforms the future state vector into its expectation condition on the states and actions indexed by I^a.

The maximization step in the dynamic programming algorithm uses a full EV evaluation:

$$I_i^a = \underset{j:\, I_x(j)=i}{\operatorname{argmax}}\, R_j + \delta[v(V)]_j$$

whereas the value function updates use an indexed evaluation. If function iteration is used

$$V \leftarrow R[I^a] + \delta v(V, I^a)$$

If policy iteration is used then V solves the linear equation:

$$h(V) = V - \delta v(V, I^a) = R[I^a]$$

Note that this linear solve cannot be performed using direct methods (e.g., LU decomposition) because the matrix operator is not available but can be solved efficiently using iterative Krylov methods.[3] Thus both standard methods for solving DP problems are still available when EV functions are used.

There are at least two situations in which the use of EV functions is advantageous. The first situation in which large gains are possible with an EV function approach arises when the state transition occurs in phases, $P = P_2 P_1$, where the transition matrix for each phase, P_i, is sparse. Typically P will be far less sparse than its components, in which this case it is possible that $P_1^\top(P_2^\top V)$ can be computed far

[3] The implicit matrix involved here, $I - \delta P^\top$, is easily shown to be row-wise strictly diagonally dominant, which is a typical sufficient condition for ensuring that an iterative linear solver converges.

faster than $P^{\top}V$ and use far less memory. This will be illustrated with a Stochastic Patch Occupancy Model (SPOM) and with a model in which, in the first stage, the action transforms the state deterministically.

The second situation is when two or more sets of the state variables have transition probabilities that are conditionally independent, where conditioning is on subsets of the current state and action variables. Such a situation arises in many dynamic programming models. This is illustrated with a harvest management example and with an SPOM model defined on a network of interconnected sites. To facilitate the specification of such EV functions a set of procedures was developed that allows a user to pass a set of transition matrices for individual state variables, along with information on the conditioning variables involved.

Staged Transitions

The first situation in which there are gains from using the EV function approach arises when the transition can be broken into separate phases, each of which can be described by a sparse transition matrix. Such a situation arises with so-called Stochastic Patch Occupancy Models (SPOMs). Early contributors to this literature are Caswell & Etter (1993), Hanski (1994) and Day & Possingham (1995). In these models there are N sites or patches that can each be classified as either empty or occupied. In one of the phases, the extinction phase, an occupied patch might change to empty with probability e (and if empty it remains so). In the other phase, the colonization phase, an empty patch might change to occupied with probability c (and if occupied it remains so). Typically e and c may differ from patch to patch and will be functions of the current condition of the other patches and of actions that resource managers take.

In SPOMs the state variable is a vector of N 0s and 1s representing the occupancy status of each patch. The number of possible configurations is 2^N which clearly is a manifestation of the curse of dimensionality. The larger issue for such models, however, is that P has 4^N elements (for any given treatment) and is typically dense or nearly so. The transition matrix however can be decomposed into its extinction and colonization phases, either as $P = EC$ or $P = CE$ where E and C represent the transition probability matrices for the two phases (which order is used depends on when action is taken). For an individual site the site transition matrices for each stage are triangular:

$$E_i = \begin{bmatrix} 1 & e_i \\ 0 & 1 - e_i \end{bmatrix} \qquad C_i = \begin{bmatrix} 1 - c_i & 0 \\ c_i & 1 \end{bmatrix}$$

Note that, in this simple model, the colonization probabilities do not depend on the occupancy status of other patches. The full extinction and colonization transition matrices can therefore be written as a sequence of Kronecker products, e.g., $E = E_1 \otimes E_2 \otimes \ldots \otimes E_N$, implying that there are 3^N non-zero values in each of E and C. (this assumes that none of the values of the e_i and c_i are exactly 0 or exactly 1), The density of these matrices is thus of $3^N/4^N$ (their sparsity pattern is shown in Figure 1 for $N = 10$). Although still problematic, storing 3^N elements in each of two sparse matrices may be feasible for values of N for which storing a dense matrix with 4^N elements is not. Also performing 3^N arithmetic operations twice is much faster than performing 4^N operations once.

These results are even more dramatic if each site can be classified into more than 2 categories. If there are m possible categories then there will be m^N values of the state and the transition matrix will contain m^{2N} values. If the two phases represent a decreasing and an increasing phase the single site phase transition matrices will be triangular and thus contain $m(m+1)/2$ non-zero elements. The number of non-zero elements in full phase transition matrix is this number raised to the power N which implies that the density of the phase transition matrix is $\left(\frac{m+1}{2m}\right)^N$. The density therefore declines towards 2^{-N} as m gets large. Clearly the curse of dimensionality is still present but at least some of its sting has been reduced.

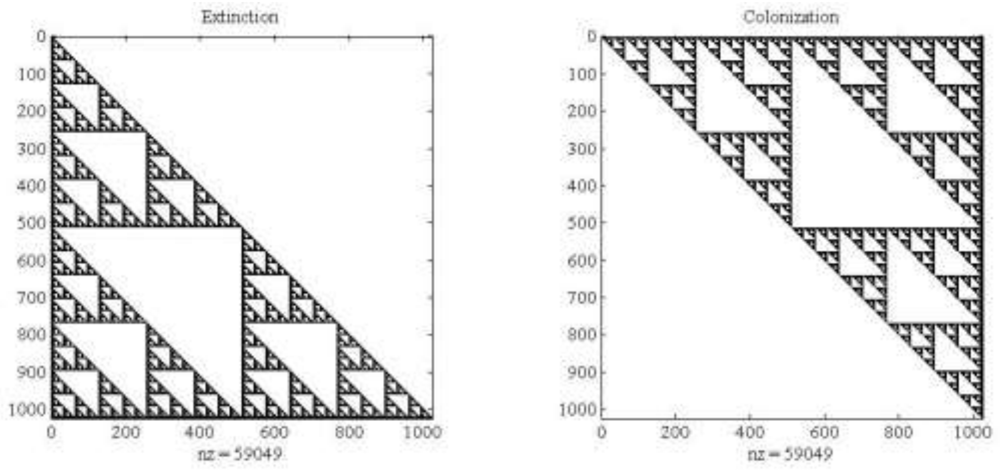


Figure 1. Sparsity patterns for extinction and colonization transition matrices ($N = 10$)

Table 1 displays the relative times required to do a basic matrix-vector multiplication, which is the basis for Krylov methods, using the full and staged transition approaches. Row 1 displays the time required for 1000 of these operations using the staged form $E^{\top}(C^{\top}V)$ and row 2 shows the same for the full form $P^{\top}V$. At relatively low values of N the full method actually is faster than the staged form, a result that is likely due to the greater efficiency of the matrix multiply operation for full versus sparse formats (this is, of course, dependent on both the software and hardware used). Once N is greater than 10, however, the staged form is faster by an increasingly wide gap, being over 13 times faster for $N = 14$. The third row of the table shows the time required to actually form P by multiplying C and E. This also imposes a significant and avoidable computational burden both in time and memory utilization.

Table 1. Typical computational times and sparsity for SPOM model

	N						
	8	9	10	11	12	13	14
$E^{\top}(C^{\top}v)$	0.026	0.065	0.086	0.136	0.292	1.672	4.870
Pv	0.014	0.036	0.084	0.801	4.011	15.298	64.277
$P = CE$	0.008	0.008	0.046	0.154	0.724	3.499	19.332
density	0.100	0.075	0.056	0.042	0.032	0.024	0.018

Rows 1 & 2 display the time required for 1000 evaluations using the factored form $E^{\top}(C^{\top}v)$ and full form $P^{\top}v$

Row 3 shows the setup time required to a form P

Row 4 shows the fraction of non-zero elements in E and C

Another way that staged transitions can lead to substantial computational gains arises when the state transition can be written in terms of the so-called post-decision state. For example, in some fisheries models the future state depends only on escapement which equals the current stock less that harvest. In a simple model the current stock is the state, the harvest is the action and the escapement is the post-harvest state.

In general if the transition can be divided into a deterministic transition $\tilde{S} = g_1(S, A)$ and a stochastic transition $S^{+} = g_2(\tilde{S}, e)$ then we only require an $n_s \times n_s$ transition matrix P_2 and an n_x index vector I_1 that defines the g_1 mapping. The expected value function can then be written as $v(V) = [P_2^{\top}V](I_1)$.

Conditional Independence

Many dynamic models consist of a d_s-element set of state variables that evolve independently when conditioned on the current state and action variables. The values of the conditioning variables can organized into an $n_x \times d_x$ matrix X, with each row representing a unique combination of states and actions. In addition to X a model is defined by a set of d_s conditional probability tables (CPTs), P_i, representing the transition probability conditioned on a subset of X and an associated set of index vectors

q_i defining the sets of conditioning (parent) variables, with the values of the q_i associated with columns of X.

The simplest case arises when the state variables have disjoint conditioning sets ($q_i \cap q_j = \emptyset$ for $i \neq j$). In this case the transition matrix can be written as a chain of Kronecker products:

$$P = P_1 \otimes \ldots \otimes P_{d_s}$$

(this was true of the SPOM discussed in the previous section). It is well known that Kronecker product-vector multiplication can be performed efficiently without actually forming the Kronecker product (Pereyra and Scherer; 1973). The model of dynamic reserve site selection of Costello and Polasky (2004) and the harvest management example discussed below both fit this framework.

In the more general case, in which the conditioning sets overlap, an EV function can be evaluated by processing each CPT sequentially using index vectors to define the associated conditioning variables. The basic approach uses a special indexed multiplication of a 3-D array by a 2-D array:

$$y_i(:,k) = \ y_{i-1}\big(:,:,I_i^y(k)\big)P_i\big(:,I_i^p(k)\ \big)$$

where I_i^y and I_i^p are index vectors that indicate the page (the 3rd dimension) of y_{i-1} and the column of P_i associated with column k of y_i. Each column of the output y_i is computed as multiplication of an $\left(\prod_{j=i+1}^{d} n_j\right) \times n_i$ matrix by an n_i vector. At each step the result y_i is reshaped in a 3-D array with n_i elements in its 2nd dimension. The process is initialized by combining V with P_1 to form y_1. The algorithm, which is discussed in greater detail in Supplemental Appendix 2, has the significant advantages that no copying or shuffling of values in memory is required and that the bulk of the work is performed using matrix-vector multiplication, which can be implemented in a highly efficient way and uses minimal memory resources.

The number of arithmetic operations is $\sum_{i=1}^{d} \prod_{j=i}^{d} n_j \min(k_i, n_s)$ (recall that $n_s = \prod_{j=1}^{d} n_i$). Contrast this with an indexed operation using $P[:, I^a]$ which uses n_s^2 arithmetic operations.

To illustrate the operations involved consider a problem with 3 state variables and 1 action variable. The state variable sizes are all n and the action has size n_a. With the action in the last column of X suppose that the parents vectors are given by

$$q_1 = [1\ \ 4] \qquad q_2 = [1\ \ 2\ \ 4] \qquad q_3 = [2\ \ 3\ \ 4]$$

So future state 1 depends on current state 1 and the action, etc. The EV function is performed in 3 steps each involving the current intermediate product y_{i-1} and the current CPT P_i. The variables involved with each array and the number of arithmetic operations required by the indexed multiplication are:

i	y_{i-1}	P_i	# of operations
1	$S_3^+ S_2^+ S_1^+$	$S_1^+ S_1 A$	$n^4 n_a$
2	$S_3^+ S_2^+ S_1 S_2 A$	$S_2^+ S_1 S_2 A$	$n^4 n_a$
3	$S_3^+ S_1 S_2 A$	$S_3^+ S_2 S_3 A$	$n^4 n_a$

The total operation count is $3n^4 n_a$. If the full transition matrix is used the operations count is $n^6 n_a$.

EV functions can be evaluated using this approach for both full evaluations of the form $v(V)$ and indexed evaluations of the form $v(V, I^a)$ where I^a is an index vector specifying a strategy. . The latter form is a bit more complicated to implement and is discussed in detail with an example in Supplemental Appendix 2.

The efficiency of computing an EV function can be influenced both by the sequencing of the state variables and by performing a preprocessing step in which some of the CPTs are combined to reduce the amount of computation performed. Determining the optimal sequencing is a difficult problem to solve and there do not appear to be any polynomial algorithms to solve it. The minimal arithmetic operation preprocessing of CPTs into groups, however, can be determined using a simple algorithm; this is discussed in detail in Supplemental Appendix 3.

To illustrate the advantage of combining CPTs in a preprocessing step consider 2 CPTs with the same conditioning sets: $q_1 = [1\ \ 2\ \ 4]$ and $q_2 = [1\ \ 2\ \ 4]$**.** The first two steps with P_1 and P_2 have operation counts

i	y_i	P_i	# of operations
1	$S_3^+ S_2^+ S_1^+$	$S_1^+ S_1 S_2 A$	$n^5 n_a$
2	$S_3^+ S_2^+ S_1 S_2 A$	$S_2^+ S_1 S_2 A$	$n^4 n_a$

If we combine P_1 and P_2 in a preprocessing step to form P_{12} the same operation has

i	y_i	P_{12}	# of operations
1	$S_3^+ S_2^+ S_1^+$	$S_2^+ S_1^+ S_1 S_2 A$	$n^5 n_a$

Thus we can do both operations in a single step with the same operation count as the previous first step.

Example: Harvest Management

To demonstrate the extent of the gains consider first the case of managing the harvest of a wild stock, such as a fishery. Models of this sort go back at least to Clarke & Munro (1975) and many variants have appeared using both continuous and discrete time formulations. Here we use a fairly simple variant in which a biological population is commercially harvested with a transition function that can be written as

$$N^+ = f(N, H)u$$

where N is the population size, H is the harvest size and u is a random noise term. Suppose that this is discretized with sorted sets of n_N values of N and n_H values of H. The resulting transition matrix P_N is $n_N \times n_N n_H$ (this can be viewed as an $1 \times n_H$ vector composed of blocks of size $n_N \times n_N$). In addition the price received (M) for the harvest evolves dynamically according to

$$M^+ = g(M)w$$

where w is also a random noise term. Proceeding as before this is discretized and the $n_M \times n_M$ transition matrix P_M is formed.

This is an example in which the conditioning sets (parent variables) form non-overlapping sets and so the transition matrix can be written as a Kronecker product. If the variables are organized lexicographically and ordered as (H, N, M) then the combined transition matrix can be written as $P = P_N \otimes P_M$. Rather than using $(P_N \otimes P_M)^\top V$ to compute the EV function we can use $P_M^\top \breve{V} P_N$ where $\breve{V}$ is the $n_m \times n_n$ matrix such that $\mathrm{vec}(\breve{V}) = V$. This expression can be computed as either $P_M^\top(\breve{V}P_N)$ or $(P_M^\top\breve{V})P_N$. The former approach requiring $n_m n_n^2 n_a + n_m^2 n_n n_a$ arithmetic operations and the latter requiring $n_m^2 n_n + n_m n_n^2 n_a$; the latter expression therefore unambiguously requires less computational effort.

This model was implemented and solved using $n_H = 51$, $n_N = 101$ and $n_M = 101$. The transitions were discretized using linear interpolation weights and either 10000 randomly generated values of u and w (Monte Carlo method) or 21 Gaussian quadrature nodes and weights (quadrature method). The dynamic programming problem was then solved using the full transition matrix with both a direct (LU) linear solver and an iterative Krylov solver (stabilized bi-conjugate gradient) and with 2 EV functions that differed in the order of operations. Using a direct solver required only 6 iterations whereas the use of the Krylov method required 10 iterations (this was true for both discretization methods). The Krylov method typically requires more iterations because is does not attempt to obtain more accuracy than is necessary at each iteration. The optimal decision strategy did not differ between the two linear solve methods.

Typical timing results are shown in Table 2. Comparison of the direct and Krylov methods using the full transition matrix (in the first two columns of numbers) clearly demonstrates the advantages possible using Krylov methods rather than direct methods with policy iteration, as has already been demonstrated by Mrkaic (2002). The further advantage of using an EV function is also demonstrated with the better of the two EV functions solving the model approximately 10 times as quickly using Krylov with the full transition matrix and 37-58 times faster than if a direct method is used. The difference in timing results for the two EV functions methods results because the second method performs the first multiplication with P_M which is much smaller than P_N.

The differences in the results for the Monte Carlo and the quadrature based methods can be explained by the differences in the degree of sparsity of the transition matrices that the 2 methods produced. P_N and P_M 12% and 39% dense with the Monte Carlo based approach and 20% and 35% with the quadrature based approach; these values imply densities of 4% and 7% for the full transition matrix. This leads to a moderate increase in time for the Krylov methods (which rely on simple matrix-vector operations) and a fairly dramatic increase in time for the direct methods. These results are, of course, specific to the particular example used here and don't allow the conclusion that the Monte Carlo approach to discretization should be preferred. Indeed initial computation of the P_N matrices differed dramatically for the two approaches (3 seconds for the quadrature versus 17 seconds for the Monte Carlo approach).

Table 2. Typical timing results for the harvest management example

	solution method			
discretization approach	full - direct	full - Krylov	$\boldsymbol{P}_{\boldsymbol{M}}^{\top}(\breve{\boldsymbol{V}}\boldsymbol{P}_{\boldsymbol{N}})$	$(\boldsymbol{P}_{\boldsymbol{M}}^{\top}\breve{\boldsymbol{V}})\boldsymbol{P}_{\boldsymbol{N}}$
Monte Carlo	25.76	6.51	2.01	0.69
quadrature	54.73	10.28	2.26	0.95

Example: Controlling a spatial network

Chadès et al. (2011) developed a Stochastic Patch Occupancy Model (SPOM) for managing networks of spatial sites that consisted of N sites with an $N \times N$ adjacency matrix C ($C_{ij} = 1$ is sites i and j are neighbors and 0 otherwise). Each site is either occupied or empty and either treated or not treated: O/T, O/N, E/T or E/N and a single site can be treated each period.

The transition probability for site i depends on whether it is occupied or empty (S_i), treated or not treated (A_i) and, if empty & not treated, on the # of occupied/untreated neighbors: $q_i = \sum_{j=1}^{N} C_{ij} S_j\,(1 - A_j)$. The transition matrix for site i can be represented by a $2 \times (4 + K_i)$ matrix

$$P_i = \begin{bmatrix} p_{ot} & p_{on} & p_{et} & p_{en}^0 & p_{en}^1 & \cdots & p_{en}^{K_i} \\ 1 - p_{ot} & 1 - p_{on} & 1 - p_{et} & 1 - p_{en}^0 & 1 - p_{en}^1 & \ldots & 1 - p_{en}^{K_i} \end{bmatrix}$$

where the probabilities of occupancy in the next period are p_{ot} (occupied, not treated), p_{on} (occupied, treated), p_{et} (empty, treated) and p_{en}^{j} (empty, untreated with j occupied/untreated neighbors, up to K_i).

The state space has size 2^N and there are $N+1$ possible actions (including doing nothing). There are, therefore, $(N+1)2^N$ state/action combinations

If EV functions are used the operation count depends on the density of the network, which can range from all isolated (no neighbors) to all connected, with the operation count increasing as the network becomes more connected. Figure 2 shows the $\log_{10}$ operation count for both isolated and fully connected networks using the EV function approach and compares this to the operation count using the full transition matrix. Even a fully connected network requires significantly fewer operations than using P; with $N = 16$ there are nearly 3 orders of magnitude fewer operations using the EV function approach.

It might seem that, for a fully connected network, there would be no advantage to using an EV function because the transition for each site depends, in principle, on the current state of every other site. In this model, however, the transition for any specific site depends only on how many of its neighbors are occupied. This means that the intermediate factors (the y_i) do not need to grow as fast as they would if the transitions depended on the identities of the occupied neighbors.

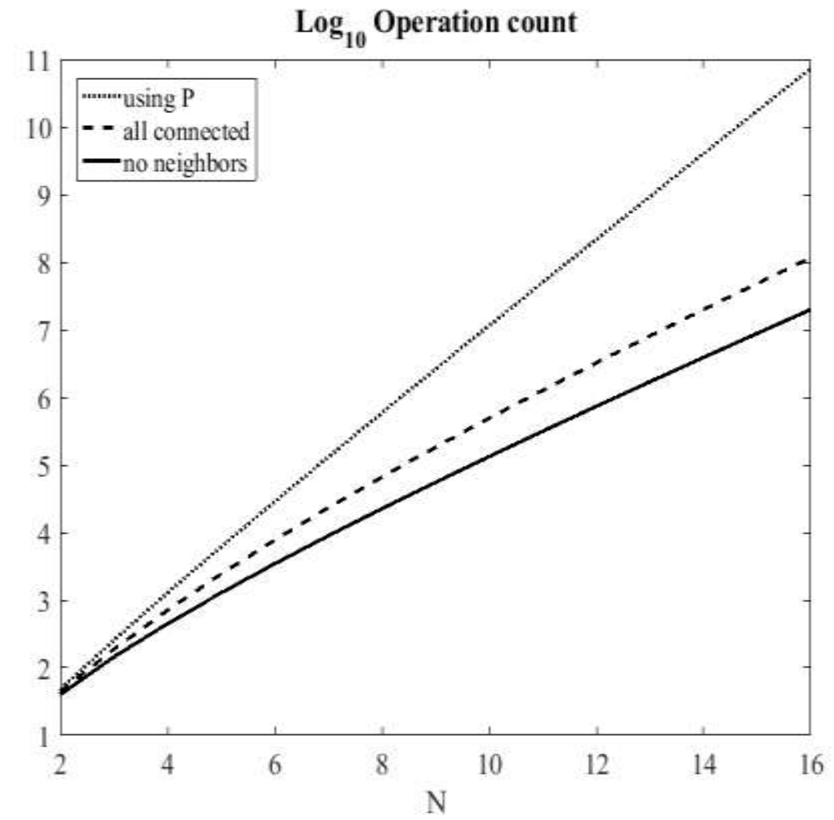


Figure 2. Operation count for spatial network model as a function of the number of sites. EV functions are used for the “no neighbors” and “all connected cases.” (SpatNet.m)

<u>Concluding comments</u>

This paper introduces the use of expected value (EV) functions as a way to at least partially address curse of dimensionality issues. Although model size still exhibits exponential grow as the number of model variables grows, the use of EV models can nonetheless make feasible the solution of models that might otherwise be out of reach and speed up the solution of models that might previously have been frustratingly slow to solve. This was demonstrated for situations for which the state transition can be broken into separate phases and transitions that can be modeled in factored form.

An important challenge for making such an approach more widely used is to recognize when these methods are applicable. Ideally this could be done by the computer so users would not have to engage in complicated programming. In some cases, such as transitions that can be broken into stages, the use of EV functions is fairly natural. It may also be easy to determine if a model can be described in factored form with the state transitions conditioned on subsets of current states and actions. In this case easy-to-use software for creating the EV function has been incorporated into the MDPSolve package. This consists of a function that accepts as inputs the CPTs (P_i), the set of parent variables for each future state variable (q_i) and the matrix of conditioning variables (X) and returns an EV function which can then be passed to the dynamic programming solver.

The examples provided here do not cover all of the possible cases for which EV functions may be useful. An important omission is one in which the CPTs for the future state variables are conditioned on noise terms that are common to 2 or more states. Such a noise term cannot be eliminated until all the state variables that it affects are already processed. This typically results in larger intermediate factors, thereby increasing both processing time and memory usage. Nonetheless, a factored approach may still improve on the use of the full transition matrix, especially if there are subsets of state variables which involve nearly disjoint sets of conditioning variables.

References

Barrett, R., M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Van der Vorst. 1994. “Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.” SIAM, Philadelphia, PA.

Caswell, H. & R.J. Etter. 1993. “Ecological interactions in patchy environments, from patch occupancy models to cellular automata.” *Lect. Notes Biomath.*, 96: 93–109.

Chadès, Iadine, Tara G. Martin, Samuel Nicol, Mark A. Burgman, Hugh P. Possingham, and Yvonne M. Buckley. 2011. “General rules for managing and surveying networks of pests, diseases, and endangered species.” *PNAS*, 108 (20) 8323-8328. https://doi.org/10.1073/pnas.1016846108

Clark, C. W. & G. R. Munro. 1975. “The economics of fishing and modern capital theory: A simplified approach.” *Journal of Environmental Economics and Management*, 2: 92–106.

Costello, Christopher and Stephen Polasky. 2004. “Dynamic Reserve Site Selection.” *Resource and Energy Economics*, 26(2): 157-174.

Day, J. & H.P. Possingham. 1995. A stochastic metapopulation model with variability in patch size and position. *Theor. Popul. Biol.*, 48: 333–360.

Fackler, Paul L. 2011. “MDPSolve User’s Guide.” Available at: https://sites.google.com/site/mdpsolve/

Hanski, I., 1994. “A practical model of metapopulation dynamics.” *J. Anim. Ecol.*, 63: 151–162.

Marescot, Lucile, Guillaume Chapron, Iadine Chadès, Paul L. Fackler, Christophe Duchamp, Eric Marboutin & Olivier Gimenez. 2013. Complex decisions made simple: A primer on stochastic dynamic programming. *Methods in Ecology and Evolution*, 4: 872–884

Mrkaic, Mico. 2002. “Policy Iteration Accelerated with Krylov Methods.” *Journal of Economic Dynamics and Control* 26: 517-45.

Pereyra, V. & Scherer, G. 1973. Efficient computer manipulation of tensor products with applications to multidimensional approximation. ACM Transactions Math. Comput., 27: 595-605.

Powell, Warren B. and Huseyin Topaloglu, 2005. Approximate Dynamic Programming for Large-Scale Resource Allocation Problems. Tutorials in Operations Research, Chapter X, INFORMS—New Orleans 2005. doi 10.1287/educ.1053.0000

Puterman, M.L. 1994 Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, New York.

Rust, John. 1996. “Numerical Dynamic Programming in Economics” in H. Amman, D. Kendrick and J. Rust (eds.) Handbook of Computational Economics. Elsevier, North Holland.

Rust, John. 2008. “Dynamic programming” in Steven N. Durlauf and Lawrence E. Blume (eds.), The New Palgrave Dictionary of Economics. Second Edition. Palgrave Macmillan.

Supplemental Appendix 1:
Index Vectors

Index vectors are vectors composed of positive integers and can be used for extraction, expansion and shuffling operations. They are used extensively in matrix based programming environments such as MATLAB and R. To illustrate let:

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 2 & 0 \\ 2 & 1 \\ 3 & 0 \\ 3 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 0 \\ 2 & 0 & 1 \\ 2 & 1 & 0 \\ 2 & 1 & 1 \\ 3 & 0 & 0 \\ 3 & 0 & 1 \\ 3 & 1 & 0 \\ 3 & 1 & 1 \end{bmatrix}$$

The index vector $I = [5 \quad 6 \quad 7 \quad 8]$ extracts the rows of B with the first column equal to 2 so $B\left(I_j, 1\right) = 2$ for every j. The index vector $I = [1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 4 \quad 4 \quad 5 \quad 5 \quad 6 \quad 6]$ expands A so $A(I,:) = B(:,[1\ 2])$. Similarly $I = [1 \quad 2 \quad 1 \quad 2 \quad 3 \quad 4 \quad 3 \quad 4 \quad 5 \quad 6 \quad 5 \quad 6]$ expands A so $A(I,:) = B(:,[1\ 3])$. Finally the index vector $I = [1 \quad 3 \quad 5 \quad 2 \quad 4 \quad 6]$ shuffles the rows of A so they are sorted by the second column rather than the first:

$$A(I,:) = \begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 3 & 0 \\ 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix}$$

Dynamic programming algorithms can be described in terms of index vectors. Consider a DP model with 2 state variables, each binary, and 3 possible actions

The matrix S lists all possible states and X lists all possible state/action combinations:

$$S = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad X = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 2 & 0 & 0 \\ 2 & 0 & 1 \\ 2 & 1 & 0 \\ 2 & 1 & 1 \\ 3 & 0 & 0 \\ 3 & 0 & 1 \\ 3 & 1 & 0 \\ 3 & 1 & 1 \end{bmatrix}$$

(note that column 1 of X is the action and columns 2 and 3 are the 2 states). The expansion index vector that gives the states in each row of X is

$I_x = [1 \quad 2 \quad 3 \quad 4 \quad 1 \quad 2 \quad 3 \quad 4 \quad 1 \quad 2 \quad 3 \quad 4]$

This expands S so $S(I_x,:) = X(:,[2\ \ 3])$.

A state dependent strategy can be specified as an extraction index vector with the ith element associated with state i:

$I^a = [1\ \ 6\ \ 7\ \ 12\ ]$ yields:

$$X(I^a,:) = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 1 \\ 2 & 1 & 0 \\ 3 & 1 & 1 \end{bmatrix}$$

i.e., a strategy that associates action 1 with state 1, action 2 with states 2 and 3 and action 3 with state 4

Strategy vectors select a single row of X for each state so $X(I^a, J^s) = S$ where J^s is an index of the columns of X associated with the state variables.

Supplemental Appendix 2:
Computational approach to evaluating EV functions

A factored model is defined by a set of d_s conditional transition probability matrices P_i of size $n_i \times m_i$. The computations necessary to compute an EV function can be implemented in a set of d_s multiplication operations involving the CPTs. The multiplication operations have a special form which can be called indexed multiplications. These involve a 3-D array X multiplied by a 2-D array Y with the arrays matched according to 2 index vectors, I^x and I^y, both of length K.

The indexed multiplication can be described as follows. Let the inputs X be $m \times n \times p$ and Y be $n \times q$ and the output Z be $m \times K$, where $Z_{:k} = X_{::I_k^x} Y_{:I_k^y}$ (the : indicates all elements for a given dimension). Thus each column of the output Z is computed as an ordinary matrix-vector product of one of the pages (3rd dimension) of X and one of the columns of Y. Note that when arrays are stored in column-major form (as is true with MATLAB) the subarrays used in the matrix-vector products are stored in contiguous memory. These matrix-vector products can be computed efficiently with a call to the BLAS gemv procedure (Netlib, BLAS (Basic Linear Algebra Subprograms), https://www.netlib.org/blas/). Let this function be represented as $Z = IM(X, I_x, Y, I_y)$. To avoid unnecessary indexing, if the index vector for either X or Y is null (empty) then the index is assumed to equal 1 through K.

The algorithm for computing an EV function can now be described. First, set $y_0 = V$ and let y_i be the intermediate product after incorporating the first i CPTs. Let I_i^p and I_i^y be index vectors with length $k_i = \prod_{j \in Q_i} n_j$ where $Q_i = \bigcup_{k=1}^{i} q_k$. In words, k_i is the size of the space of conditioning variables for the first i state variables.

Using the I index vectors a full EV function evaluation is computed using the following pseudo-code:

```
set $y = v$
reshape $y$ to be $\prod_{j=2}^{d} n_j \times n_1$
set $y \leftarrow y * p_1$
loop from $i = 2$ to $i = d$
   reshape $y$ to be $\left(\prod_{j=i+1}^{d} n_j\right) \times n_i \times k_{i-1}$
   set $y \leftarrow IM(y, I_i^y, P_i, I_i^p)$
return $y$
```

The total operation count is $\sum_{i=1}^{d} p_i k_i$ where $p_i = \prod_{j=i}^{d} n_j$ is the size of the space of the remaining unprocessed state variables. This can be contrasted to the use of the full transition matrix, which uses $n_s n_x$ operations. Note that variable order matters and ideally we want the k_i to grow slowly. It should also be noted that the reshape operation that transforms a $\left(\prod_{j=i}^{d} n_j\right) \times k_{i-1}$ matrix into a $\left(\prod_{j=i+1}^{d} n_j\right) \times n_i \times k_{i-1}$ 3-D array has no computational cost as it does not require access to the elements of the array but merely alters how those elements are interpreted.

The discussion thus far has applied to a full EV evaluation which returns $E[V(S^+)|X]$ for all state/action combinations. When the dynamic programming algorithm is carried out using policy iteration and Krylov methods most EV evaluations are indexed. Hence we also require an efficient way to compute $E[V(S^+)|X]$ for a specific strategy. A strategy can be defined by the index vector I^a (with length n_s). Although it is possible to simply do a full (non-indexed) evaluation and then extract the elements using I^a such an approach would perform a large amount of unnecessary computations.

An alternative uses a set of J_i^p index vectors that expand the columns of P_i to match those of the full X matrix. Each J_i^p is a vector of length n_x (i.e., equals the # of rows of X). The algorithm could be initialized as before ($y \leftarrow y * p_1$) and then y could be expanded by setting ($y \leftarrow y(:, J_1^p)$. Then, looping over the remaining CPTS we could use $y \leftarrow IM(y, [], P_i, J_i^p)$

Where [] represents a null (empty) input. A more efficient approach recognizes that early in the operation it is generally more efficient to use the I^p indices and latter it is more efficient to use the J^p indices. At some point the length of I^p is greater than n_s (the length of I^a), at which point it would be more efficient to switch to the use of the J^p indices. To implement this we also need an additional index vector J^y to expand y_i at the time the switch is made.

The indexed EV function evaluation is described by the following pseudo-code:

```
set $y = v$
reshape $y$ to be $\prod_{j=2}^{d} n_j \times n_1$
set $y \leftarrow y * p_1$
set useI = true
loop from $i = 2$ to $i = d$
    if $m_i > n_s$
        reshape $y$ to be $\left(\prod_{j=i+1}^{d} n_j\right) \times n_i \times m_{i-1}$
            and expand $y(:,:,k) \leftarrow y\left(:,:,J^y\left(I^a(k)\right)\right)$
        set useI = false
    if useI=true
        reshape $y$ to be $\left(\prod_{j=i+1}^{d} n_j\right) \times n_i \times m_{i-1}$
        set $y \leftarrow IM(y, I_i^y, P_i, I_i^p)$
    otherwise
        set $y \leftarrow IM\left(y, [], P_i, J_i^p\left(I^a\right)\right)$
```

To illustrate the impact of this algorithm recall the numerical example given in the paper. Furthermore, suppose that $n < n_a < n^2$ and note that a strategy index has length $n_s = n^3$. The I_i indices have sizes nn_a, $n^2 n_a$ and $n^2 n_a$. The crossover from I to J indexing would therefore occur in step 2.

i	y_i	P_i	# of operations
1	$S_3^+ S_2^+ S_1^+$	$S_1^+ S_1 A$	$n^4 n_a$
2	$S_3^+ S_2^+ S_1 S_2 S_3$	$S_2^+ S_1 S_2 S_3$	n^5
3	$S_3^+ S_1 S_2 S_3$	$S_3^+ S_1 S_2 S_3$	n^4

The total operation count is $n^4(n_a + n + 1)$. If the full transition matrix is used by extracting the appropriate columns of P: $P[:, I^a]$ the operation requires n^6 operations.

Supplemental Appendix 3:
Optimal preprocessing of CPTs

It can be advantageous to preprocess groups of state variables into joint CPTs, especially when the variables in the group have similar sets of conditioning variables. The optimal grouping of operations can be solved using an $O(d^3)$ dynamic programming algorithm that is similar to the approach used to address the well-known matrix chain multiplication problem. Given a variable order the cost of incorporating a CPT that groups variables i through $j \geq i$ is $C_{ij} = p_i m_j$, where $p_i = \prod_{k=i}^{d} n_k$ and m_j is the number of tuples of the parents of variables 1 through j. For each (i, j) we can evaluate whether breaking the grouped variables into two further groups results in a less costly set of operations:

$$M_{ij} = \min\left(C_{ij}, \min_{k \in \{0,\ldots,j-i+1\}} M_{i,i+k} + M_{i+k+1,j}\right)$$

The minimal cost grouping is given by M_{1d}. This is optimal for a full evaluation. For an indexed evaluation we could instead define

$$C_{ij} = p_i \min(m_j, n_s)$$

By storing where splits occur the optimal groupings can be determined.