

Approximate Policy Iteration for Semiconductor Fab-Level Decision Making - a Case Study

Ying He yhe@isr.umd.edu
Shalabh Bhatnagar shalabh@isr.umd.edu
Michael C. Fu mfu@isr.umd.edu
Steven I. Marcus marcus@isr.umd.edu
Institute for Systems Research
University of Maryland
College Park, MD 20742
<http://www.isr.umd.edu/IPDPM/>

Abstract

In this paper, we propose an approximate policy iteration (API) algorithm for a semiconductor fab-level decision making problem. This problem is formulated as a discounted cost Markov Decision Process (MDP), and we have applied exact policy iteration to solve a simple example in prior work [1]. However, the overwhelming computational requirements of exact policy iteration prevent its application for larger problems. Approximate policy iteration overcomes this obstacle by approximating the cost-to-go using function approximation. Numerical simulation on the same example shows that the proposed API algorithm leads to a policy with cost close to that of the optimal policy.

Keywords: Approximate Policy Iteration, Semiconductor Fab-Level Decision Making, Markov Decision Processes, Discounted Cost Problem.

Introduction

The planning and scheduling of a semiconductor fab is carried out according to a general hierarchical framework based on a temporal and/or physical decomposition of the system. In our IPDPM (Integrating Product Dynamics and Process Models, <http://www.isr.umd.edu/IPDPM/>) project, we attempt to deal with decision making at the fab level, and issues that must be addressed in this level include, for example, when to add additional capacity and when to convert from one type of production to another [2].

For fab-level decision making, our approach is to formulate it as a Markov decision process (MDP) [3] by defining appropriate states, actions, transition probabilities, time horizon, and cost criterion. In prior work [1], we have provide a case study with exact policy iteration. However, the overwhelming computational requirements of exact policy iteration prevent its application for large problems. To overcome this obstacle, an approximate policy iteration (API) algorithm is proposed. Numerical simulation on the same case study as in [1] shows that the proposed API algorithm leads to a policy with cost close to that of the optimal policy.

Markov Decision Processes

A Markov Decision Process is a framework containing states, actions, costs, probabilities and the decision horizon for the problem of optimizing a stochastic discrete-time dynamic system. The dynamic system equation is

$$x_{t+1} = f_t(x_t, u_t, w_t), \quad t = 0, 1, \dots, T - 1, \quad (1)$$

where t indexes a time epoch; x_t is the state of the system; u_t is the action to be chosen at time t ; w_t is a random disturbance which is characterized by a conditional probability distribution $P(\cdot | x_t, u_t)$; and T is the decision horizon. We denote the set of possible system states by S and the set of allowable actions in state $i \in S$ by $U(i)$. We assume S , $U(i)$, and $P(\cdot | x_t, u_t)$ do not vary with t . We further assume that the sets S and $U(i)$ are finite sets, where S consists of n states denoted by $0, 1, \dots, n - 1$.

If, at some time t , the system is in state $x_t = i$ and action $u_t = u$ is applied, we incur a stage cost $g(x_t, u_t) = g(i, u)$, and the system moves to state $x_{t+1} = j$ with probability $p_{ij}(u) = P(x_{t+1} = j | x_t = i, u_t = u)$. $p_{ij}(u)$ may be given a priori or may be calculated from the system equation and the known probability distribution of the random disturbance. $g(i, u)$ is assumed bounded.

Consider the discounted cost problem, where there is a discount factor less than one. The objective is to minimize over all policies $\pi = \{\mu_0, \mu_1, \dots\}$ with $\mu_t : S \rightarrow U$, $\mu_t(i) \in U(i)$ for i and t , the total expected cost,

$$J_\pi(i) = \lim_{T \rightarrow \infty} E \left\{ \sum_{t=0}^{T-1} \alpha^t g(x_t, \mu_t(x_t)) \mid x_0 = i \right\}. \quad (2)$$

where α is the discount factor with $0 < \alpha < 1$.

A stationary policy is an admissible policy of the form $\pi = \{\mu, \mu, \dots\}$; we denote it by μ_∞ .

Under certain assumptions [4], the following hold:

- The optimal costs $J^*(0), \dots, J^*(n-1)$ satisfy optimality equations,

$$J^*(i) = \min_{u \in U(i)} [g(i, u) + \alpha \sum_{j=0}^{n-1} p_{ij}(u) J^*(j)], \quad i = 0, \dots, n-1, \quad (3)$$

and they are the unique solution of this equation.

- For any stationary policy μ_∞ , the costs $J_\mu(0), \dots, J_\mu(n-1)$ are the unique solution of the equation

$$J_\mu(i) = [g(i, \mu(i)) + \alpha \sum_{j=0}^{n-1} p_{ij}(\mu(i)) J_\mu(j)], \quad i = 0, \dots, n-1. \quad (4)$$

One method to solve the optimality equations is policy iteration. Policy iteration consists of a sequence of policy evaluation and policy improvement at each iteration. At each iteration step k , a stationary policy $\mu_\infty^k = \{\mu^k, \mu^k, \dots\}$ is given.

1. **Policy evaluation:** obtain the corresponding cost-to-go $J_{\mu^k}(i)$ satisfying (4).
2. **Policy improvement:** find a stationary policy μ^{k+1} , where for all i , $\mu^{k+1}(i)$ is such that

$$g(i, \mu^{k+1}(i)) + \alpha \sum_{j=0}^{n-1} p_{ij}(\mu^{k+1}(i)) J_{\mu^k}(j) = \min_{u \in U(i)} [g(i, u) + \alpha \sum_{j=0}^{n-1} p_{ij}(u) J_{\mu^k}(j)]. \quad (5)$$

If $J_{\mu^{k+1}} = J_{\mu^k}$ for all i , the algorithm terminates; otherwise, the process is repeated with μ^{k+1} replacing μ^k .

Under certain assumptions, the policy iteration algorithm terminates in a finite number of iterations with a stationary optimal policy.

Another method of solving the optimality equations is value iteration. It is done by using the recursion

$$J_{k+1}(i) = \min_{u \in U(i)} [g(i, u) + \sum_{j=0}^{n-1} p_{ij}(u) J_k(j)], \quad i = 0, \dots, n-1. \quad (6)$$

given any initial conditions $J_0(0), \dots, J_0(n-1)$.

Approximate Policy Iteration

Approximate policy iteration algorithms have the same structure as exact policy iteration except for two differences [5]:

- Given the current policy μ , the corresponding cost-to-go function J_μ is not computed exactly. Instead, an approximate cost-to-go function $\tilde{J}_\mu(i, r)$ is computed, where r is a vector of tunable parameters.
- Once approximate policy evaluation is completed and $\tilde{J}_\mu(i, r)$ is available, we generate a new policy $\bar{\mu}$ which is greedy with respect to \tilde{J}_μ . The greedy policy can be calculated exactly, or approximated.

From another point of view, approximate policy iteration algorithm consists of four modules [5], see Figure 1.

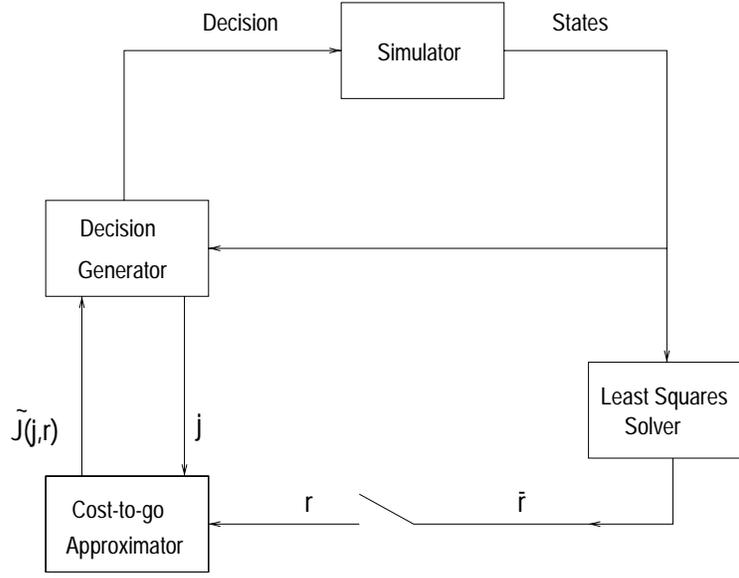


Figure 1: Structure of the Approximate Policy Iteration

- The *cost-to-go approximator*, which is the function $\tilde{J}_\mu(i, r)$;
- The *simulator*, which given a state-decision pair (i, u) , generates the next state j according to the correct transition probabilities. The m th sample path starting with state i is denoted by $c(i, m)$;
- The *least squares solver*, which accepts as input the sample paths by the simulator and solves a least squares problem

$$\min_r \sum_i \sum_{m=1}^{M(i)} (\tilde{J}(i, r) - c(i, m))^2$$

to obtain the parameter \bar{r} for calculating the approximation $\tilde{J}_\mu(i, \bar{r})$ of the cost function.

- The *decision generator*, which generates the decision $\bar{\mu}(i)$ of the improved policy at the current state i .

$$\bar{\mu}(i) = \arg \min_{u \in U(i)} \sum_{j=0}^n p_{ij}(u) (g(i, u, j) + \alpha \tilde{J}(j, r)), \quad i = 0, \dots, n-1,$$

Cost-to-go Approximator

In order to develop a cost-to-go approximator, first we need to choose an *approximation architecture*, that is, a certain functional form involving a number of free parameters. Broadly, approximation architectures can be classified into two main categories: linear and nonlinear. A linear architecture is of the general form

$$\tilde{J}(i, r) = \sum_{k=0}^K r(k) \phi_k(i),$$

where $r(k), k = 1, \dots, K$, are the components of the parameter vector r , and ϕ_k are fixed, easily computable functions. A common nonlinear architecture is a multilayer perceptron with a single hidden layer which has the following form

$$\tilde{J}(i, r) = \sum_{k=0}^K r(k) \sigma \left(\sum_{l=1}^L r(k, l) x_l(i) \right),$$

where $r(k, l)$ and $r(k)$ are coefficients for the input layer and output layer, respectively; $\sigma(\cdot)$ is a sigmoidal function such as the logistic function

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}}$$

It is often the case that the approximation architecture is too complicated for state representation and one considers the use of some structural pieces to represent states. These structural pieces are called *features*, which are fed into the approximation architecture instead of the state itself. Usually, these features are handcrafted, based on the particular problem. Some example of features include state variables, heuristic cost-to-go and/or past cost-to-go etc.

Simulator

In order to design a good simulator, we first need to define and validate a simulation model, which should contain only enough detail to capture the essence of the system for the purposes for which the model is intended. Secondly, we need to construct a computer program for the simulation model. Tasks include state transition construction, random variate generation, and verification. Thirdly, we need to design the experiment. Decisions have to be made on such issues as initial conditions for the simulation runs, the length of the simulation runs, and the number of the replications etc.

Since we are dealing with an infinite horizon problem, and simulation can only run for a finite number of steps, we need to make sure that the cost of the finite approximation is close enough to that of the infinite horizon cost. In fact, we can prove that if one sample path involves N transitions, the expected value of the resulting sample cost-to-go is within $\frac{G\alpha^N}{(1-\alpha)}$ of J^μ , where G is an upper bound on $|g(i, u, j)|$. With N set to be sufficiently large, the effects of using a finite trajectory can be made arbitrarily small.

Least Squares Solver

One set of computational methods for the least squares solver is to simulate a number of trajectories, collect the results, formulate the least squares problem and then solve it in batch mode.

Another way to solve the least square problem is incremental gradient method. Given a sample state trajectory (i_0, i_1, \dots, i_N) generated using the policy μ , the parameter vector r is updated by

$$r := r - \gamma \sum_{k=0}^{N-1} \nabla \tilde{J}(i_k, r) (\tilde{J}(i_k, r) - \sum_{m=k}^{N-1} g(i_m, \mu(i_m), i_{m+1}))$$

where γ is a stepsize. A key advantage of incremental algorithms is that they allow us to decide whether or not to simulate more trajectories, depending on the quality of the results obtained so far.

Decision Generator

If we know the exact model of the system, the decision generator can be constructed easily. If not, we also need to seek the help from simulation.

The Fab-Level Decision Making MDP Model

Model

In the fab-level decision making MDP model, the state is summarized by a vector of capacities $X(t)$ at time epoch t , where the components $X_{(l,i),w}(t)$ represent the capacity (measured, for example, in wafer starts per day or number of machines) of type w allocated to product l and operation i (this could be a type of sub-factory manufacturing a particular product or a type of process). Actions to be taken could be the decisions to

- (i) increase the capacity of type w by $B_w(t)$ units, possibly by the introduction of new technology; or
- (ii) switch over $V_w^{(l,i),(m,j)}(t)$ units of type w capacity from product l and operation i to product m and operation j (for example, by qualifying tools for a different process).

Randomness is explicitly modeled by the demand $d_l(t)$ for product l . The dynamics of the model includes the fact that, after the decision is made to increase capacity, there is a delay, possibly random, in the ability to fully utilize the increased capacity, and that the capacity may gradually ramp up to the expected level. The evaluation criteria include a number of factors, including costs for excess capacity and capacity shortages, cost of production, cost of converting capacity from one type of operation to another, and the cost of increasing capacity. A more precise description of the model is given below.

The state vector in period t (between time epoch t and time epoch $t + 1$) is given by $X(t) = (T_w(t), X_{(l,i),w}(t), I_l(t), l \in \mathcal{P}_t \setminus \{0\}, i \in w \setminus \{0\}, w \in \mathcal{A}_t)^T$. The actions vector in period t is $U(t) = (B_w(t), D_w(t), V_w^{(l,i),(m,j)}(t), w \in \mathcal{A}_t, l, m \in \mathcal{P}_t, i, j \in w, w \in \mathcal{A}_t)^T$, where it is assumed that $V_w^{(l,i),(m,j)}(t) = 0$ if $(l, i) = (m, j)$. At the beginning of any period, the decision maker observes the state of the system and chooses an action.

The total cost over an infinite planning horizon that we want to minimize is

$$J = \lim_{T \rightarrow \infty} E \left[\sum_{t=0}^{T-1} \alpha^t g(X(t), U(t)) \right]$$

where

$$\begin{aligned} g(X(t), U(t)) &= \sum_{w \in \mathcal{A}_t} (C_w^a(B_w(t)) + C_w^b(D_w(t))) \\ &+ \sum_{w \in \mathcal{A}_t} \sum_{\{(l,i),(m,j) | l, m \in \mathcal{P}_t, i, j \in w, (l,i) \neq (m,j)\}} C_w^c(V_w^{(l,i),(m,j)}(t)) \\ &+ \sum_{l \in \mathcal{P}_t} (C_l^d(I_l(t)) + \sum_{w \in \mathcal{A}_t} \sum_{\{(l,i) \in \mathcal{P}_t \times w\}} C_w^e(X_{(l,i),w}(t))) \end{aligned}$$

We have the following state equations:

$$T_w(t+1) = T_w(t) + B_w(t) - D_w(t), \quad w \in \mathcal{A}_t, \quad (7)$$

$$X_{(l,i),w}(t+1) = X_{(l,i),w}(t) + \sum_{\{(m,j) \in \mathcal{P}_t \times w \mid (m,j) \neq (l,i)\}} (V_w^{(m,j),(l,i)}(t) - V_w^{(l,i),(m,j)}(t)), \quad (8)$$

where $l \in \mathcal{P}_t$, $i \in w$, $l \neq 0$, $i \neq 0$, $w \in \mathcal{A}_t$,

$$I_l(t+1) = I_l(t) + \min_i \left\{ \sum_{\{w \in \mathcal{A}_t \mid F_{(l,i),w} > 0\}} C_{(l,i),w} X_{(l,i),w}(t) \right\} - d_l(t), \quad l \in \mathcal{P}_t \setminus \{0\}. \quad (9)$$

The second item on the right-hand side of Equation (9) is referred to as the *throughput* in the inventory equation and gives the number of “finished wafers” of product l in period t . The operation minimizing this throughput term is referred to as the *bottleneck operation* for product l in period t .

See [2] for more details, such as notations and constraints on states.

Case Study

Simple Example

We use the same simple example as in [1]. Specifically, the fab will be characterized as follows:

- two products: “A” and “B”;
- two operations on each: “litho” and “etch”, distinguished by product;
- machines – litho or etch – could be flexible (able to do the respective operation on both products A and B) or dedicated (only able to do the respective operation on one of A or B);
- operation times, which depend on the product and the machine.

We begin by defining products, operations, and machines.

Products l are chosen from the set

$$\mathcal{P}_t = \{0, A, B\},$$

where 0 corresponds to ‘no product’.

Operations i are chosen from the set $\mathcal{X}_t = \{1, 2, 3, 4\}$, $\mathcal{N}_t = 4$, where

$$\begin{aligned} 1 &\longleftrightarrow \text{lithoA,} \\ 2 &\longleftrightarrow \text{etchA,} \\ 3 &\longleftrightarrow \text{lithoB,} \\ 4 &\longleftrightarrow \text{etchB.} \end{aligned}$$

Thus, operations 1 and 2 correspond to operations on product A, whereas operations 3 and 4 correspond to operations on product B.

Machines are words w chosen from the set \mathcal{Z}_t :

- 01 = machine dedicated to lithoA operation,
- 02 = machine dedicated to etchA operation,
- 03 = machine dedicated to lithoB operation,
- 04 = machine dedicated to etchB operation,
- 013 = flexible litho machine,
- 024 = flexible etch machine,
- $\cup \{012, 034, 014, 023, 0123, 0124, 0134, 0234, 01234\}$,

where 0 corresponds to ‘no operation’ and the last set’s elements do not correspond to feasible machines in the real model.

Other model input (system) parameters that must be defined are $C_{(l,i),w}$ and $F_{(l,i),w}$, which essentially specify the operation times for a particular product on a particular machine.

$$F_{(l,i),w} = \begin{cases} 1 & (A, 1), 013; (A, 2), 024; (B, 3), 013; (B, 4), 024; (A, 1), 01; (A, 2), 02; (B, 3), 03; (B, 4), 04; \\ 0 & \text{otherwise,} \end{cases}$$

i.e., all products require a single operation on the appropriate machine.

$$C_{(l,i),w} = \begin{cases} 1 & (A, 1), 013; (A, 2), 024; \\ 0.5 & (B, 3), 013; (B, 4), 024; \\ 1.2 & (A, 1), 01; (A, 2), 02; \\ 0.6 & (B, 3), 03; (B, 4), 04; \\ 0 & \text{otherwise,} \end{cases}$$

i.e., a flexible etch or litho machine completes one operation on product A in an hour, product B takes twice as long on both operations, and a flexible machine is 20% slower than a dedicated one (e.g., 60 minutes versus 50 minutes for product A).

For simplicity, we will take $K_w = 1$ for all w , i.e., availability is 100% for all machines.

Assumptions

Following [1], the following assumptions are made:

- During the decision horizon, no machine is purchased, discarded, or sent to reserve, and no maintenance is required. The only actions are to switch flexible machines between different products.
- For each type of machine (litho or etch), no more than one machine can be switched from one product and/or operation to another in a period.
- Products A and B are operated in whole unit and half units, respectively.
- The inventory warehouses for products A and B have capacities of 1 and 0.5 units, respectively.
- There is a limit on backlogged demand of 1 and 0.5 units for product A and B, respectively. Demand exceeding backloging limits is lost.

- The demand process for a given product is independent and identically distributed from period to period, and demand processes are mutually independent between products.

States

For the special example in [1], and under the above assumptions, the state vector of our MDP model takes the form $\{(X_{(A,1),013}, X_{(A,2),024}, I_A, I_B)\}$, where the first and second components are, respectively, the litho and etch capacities allocated to product A, and the third and fourth components are, respectively, the inventory levels of products A and B. Note that the capacity allocated to product B is simply the remainder of total machine capacity for each tool type (litho or etch), because we have assumed for this simple example that no capacity is ever put into reserve, thus reducing the dimensionality of the state vector from six dimensions to four components. Under our assumptions, the components of the state vector take values in the following sets:

$$\begin{aligned} X_{(A,1),013} = 2 - X_{(B,3),013} &\in \{0, 1, 2\}, \\ X_{(A,2),024} = 2 - X_{(B,4),024} &\in \{0, 1, 2\}, \\ I_A &\in \{-1, 0, 1\}, \\ I_B &\in \{-0.5, 0, 0.5\}, \end{aligned}$$

and thus the total number of possible states is 81 and the throughput for each product is given by [1]

$$\begin{aligned} TP_A &= \min \{X_{(A,1),013}, X_{(A,2),024}\}, \\ TP_B &= 0.5 \min \{X_{(B,3),013}, X_{(B,4),024}\}, \end{aligned}$$

where the arg min gives the bottleneck operation for the product (1 or 2 for A; 3 or 4 for B). As in [1], a *state group* is defined as the set of those states that have the same capacity allocation $X_{(A,1),013}$ and $X_{(A,2),024}$, i.e., they differ only in their product inventory levels.

Actions

For the action vector, now we have two versions. Version 1 corresponds to the definition of $V_w^{(l,i),(m,j)}$ in [1], $V_w^{(l,i),(m,j)} = 0$ if some type w capacity is switched over from product m and operations of type j to product l and operations of type i , which corresponds to the action vector with four-dimensional form $(V_{013}^{(A,1),(B,3)}, V_{024}^{(A,2),(B,4)}, V_{013}^{(B,3),(A,1)}, V_{024}^{(B,4),(A,2)})$, where the 1st component is the litho capacity moved from product A to product B, the 2nd component is the etch capacity moved from product A to product B, the 3rd component is the litho capacity moved from product B to product A, and the 4th component is the etch capacity moved from product B to product A.

Given that we consider only allocation and not expansion, and do not allow capacity to be sent to reserve, the action vector can be reduced to the two-dimensional form $(V_{013}^{(A,1),(B,3)}, V_{024}^{(A,2),(B,4)})$, since the amount of capacity moved from A to B is the negative of that moved from B to A:

$$\begin{aligned} V_{013}^{(A,1),(B,3)} &= -V_{013}^{(B,3),(A,1)} \in \{-1, 0, 1\}, \\ V_{024}^{(A,2),(B,4)} &= -V_{024}^{(B,4),(A,2)} \in \{-1, 0, 1\}. \end{aligned}$$

We will label the resulting nine possible actions as follows: A1= $(-1,-1)$, A2= $(-1,0)$, A3= $(-1,+1)$, A4= $(0,+1)$, A5= $(0,0)$, A6= $(0,+1)$, A7= $(+1,-1)$, A8= $(+1,0)$, A9= $(+1,+1)$. However, note that for a given state group, not all actions are admissible.

Demand Distribution and Transition Probability

The demands for product A and B are modeled as Bernoulli process, in which demand for product A can only be 1 or 2 units and demand for product B can only be 0.5 and 1 unit. The probabilities of demand for product A to be 1 and of demand for product B to be 0.5 are denoted as p_A and p_B .

Note that for our problem, states can be divided into *state groups*, and the state group transition is deterministic, so we can structurally construct the transition probability matrix.

First, for each action, build state group transition probability matrices, which only contain zero or some symbol, since the transitions are deterministic.

Then, we need to decide what we need to fill in each element of the state group transition matrices. That is, the zero element is replaced by a zero matrix; and the symbol element is replaced by $P(A, TP_A, B, TP_B)$, where TP_A and TP_B depend on pre-action state group.

We now show how to construct $P(A, TP_A, B, TP_B)$. With regard to different throughput of product A and B, build marginal transition probability matrices (3×3) for each product A and B for different throughputs, i.e. $P(A, TP_A)$ and $P(B, TP_B)$, based on the following inventory balance equation [1]:

$$I_l(t+1) = I_l(t) + TP_l - d_l(t)$$

Then, build the joint transition probability matrix (9×9) with regard to each throughput pair:

$$P(A, TP_A, B, TP_B) = P(A, TP_A) \otimes P(B, TP_B)$$

where \otimes is the Kronecker product.

Cost Structure

Here we only consider a linear cost structure. So the inventory cost and backlogging cost are proportional to the inventory level; the operation cost is proportional to the allocated capacity for different products on different machines; the switch-over cost is proportional to the switched capacity, which is also an action variable.

Correspondingly, the parameters are unit inventory cost and backlogging cost for both products A and B; unit operating cost on the litho machine for product A, on the litho machine for product B, on the etch machine for product A, and on the etch machine for product B; and the unit switch-over cost on both litho and etch machines.

Simulation

To illustrate the effectiveness of approximate policy iteration, a case study is provided. The approximate policy iteration is implemented as follows:

- Linear architecture is chosen for cost-to-go approximator.
- Features are state variables ($\{X_{(A,1),013}, X_{(A,2),024}, I_A, I_B\}$) and their squares, plus $\phi_1(x) = 1$; there are nine features.
- For the simulator, random variates are constructed using the inverse transform method [6]; the simulation length and number of replication are chosen to be 100 and 20, balancing between accuracy and simulation speed.

- For the least squares solver, the batch mode is adopted.

The program is written in MATLAB.

The example simulated has the following specifications.

Model specifications	
$D_A = 1$	0.4
$D_A = 2$	0.6
$D_B = 0.5$	0.7
$D_B = 1$	0.3
unit inventory cost for product A	2
unit inventory cost for product B	1
unit backlog cost for product A	10
unit backlog cost for product B	5
unit operating cost on litho machine for product A	0.2
unit operating cost on litho machine for product B	0.2
unit operating cost on etch machine for product A	0.1
unit operating cost on etch machine for product B	0.1
unit switch cost on litho machine	3
unit switch cost on etch machine	3

Table 1: Model Specifications

Starting with an initial policy (see the policy in the first iteration of Fig 2, which is an arbitrarily chosen admissible policy), after 7 iterations the program terminates with a near optimal policy. See Fig 2 and Fig 3 for the policy trajectories and cost trajectories at each iteration.

Compared to exact policy iteration for the same case in Fig 4, there are several interesting points:

- Approximate policy iteration takes 7 iterations to obtain its optimal policy, whereas exact policy iteration takes only 4 iterations to obtain the optimal policy.
- The near optimal policy is different from the optimal policy. Even if we use the optimal policy as the initial policy, approximate policy iteration still converges to the near optimal policy. This is due to the feature approximation.
- Although the paths to the optimal policy are different for approximate policy iteration and policy iteration, there are some similarities. It appears that approximate policy iteration improves slower than exact policy iteration, with the same trend.
- The cost-to-go of approximate policy iteration stalls at some policy for three steps, which cause the delay.

Conclusions and Future Works

In this paper, we present a case study of fab-level decision making using approximate policy iteration. The purpose of this work is to find ways to overcome the obstacle induced by heavy numerical computation using exact policy iteration.

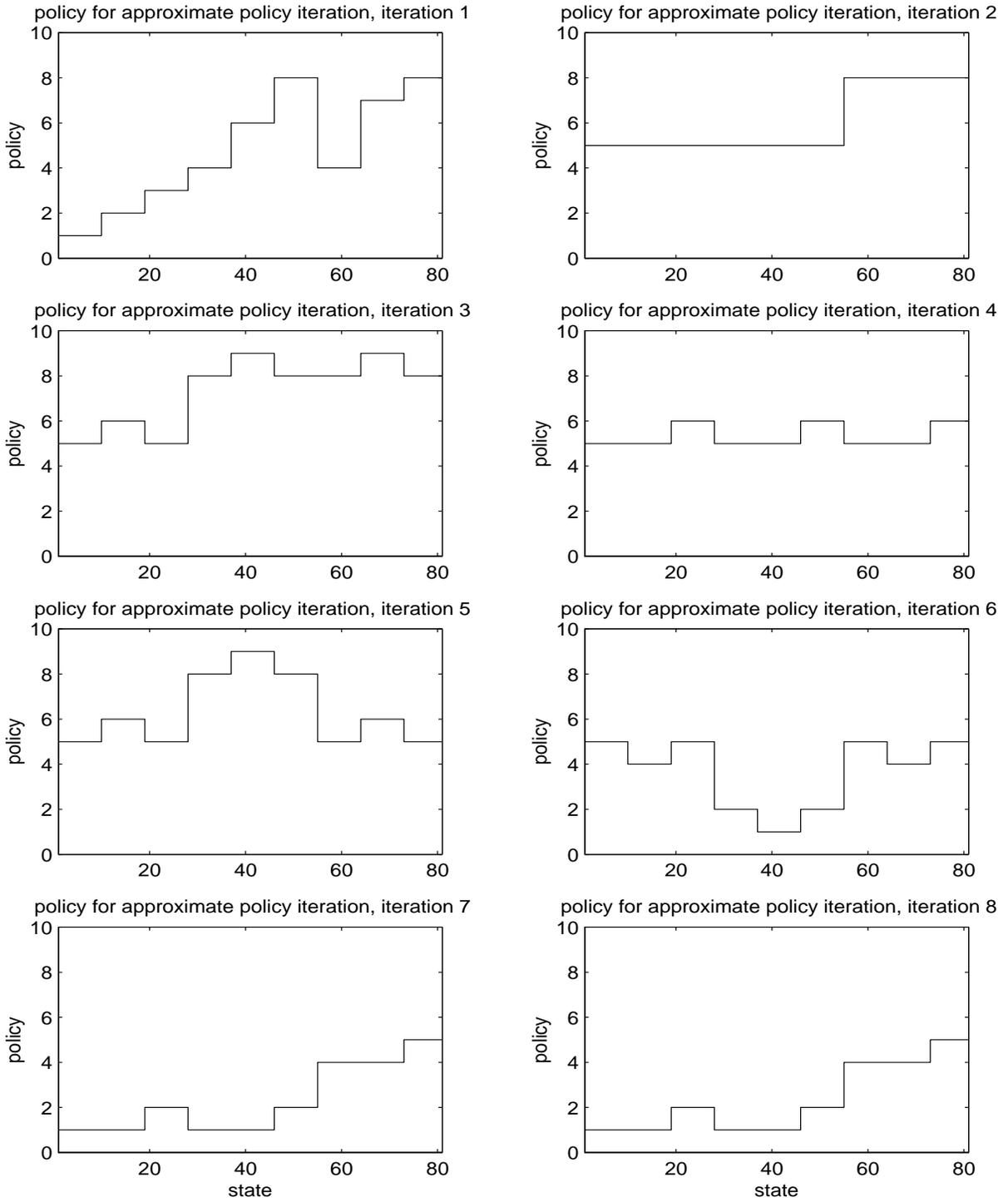


Figure 2: policy for approximate policy iteration

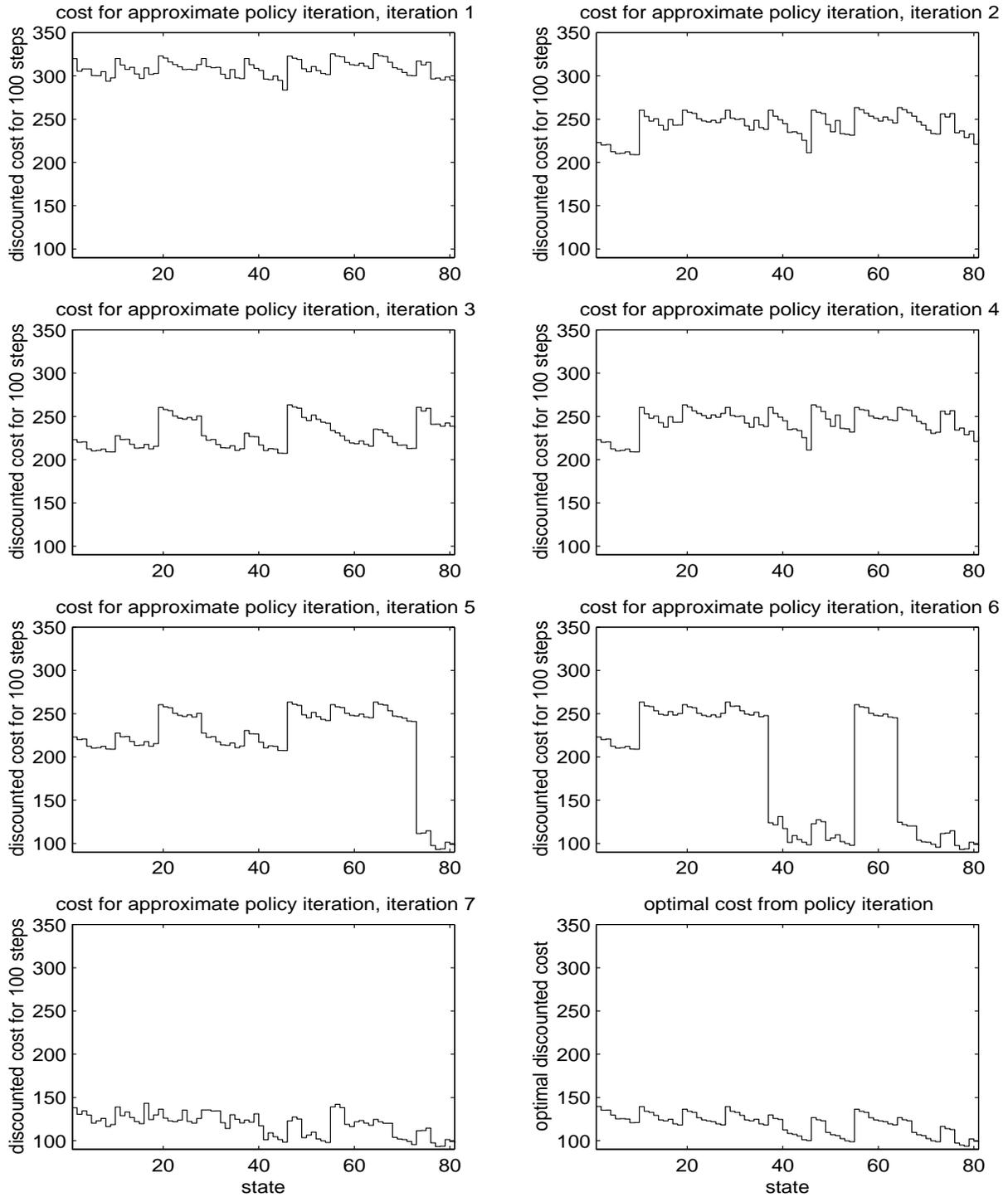


Figure 3: cost function for approximate policy iteration

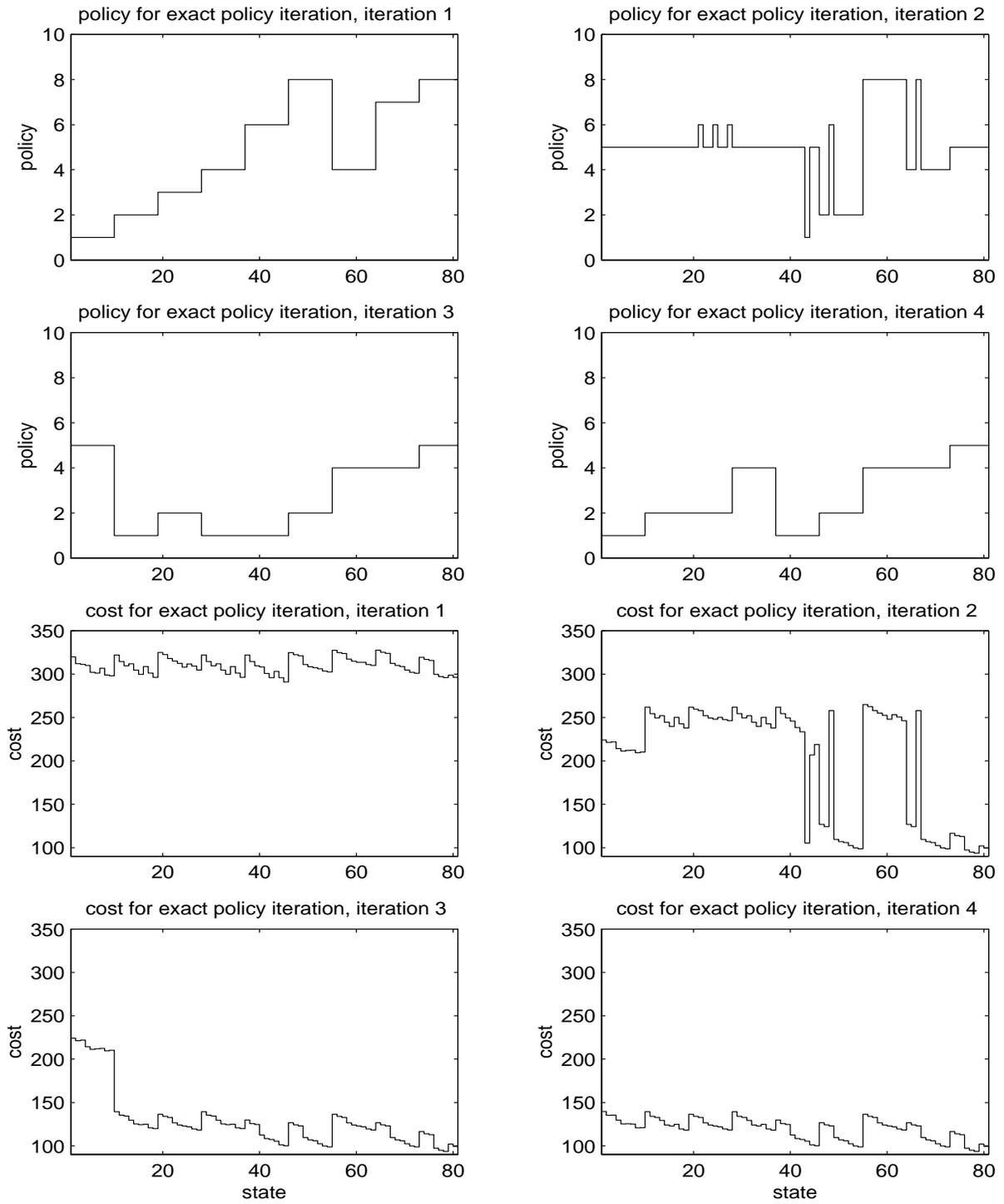


Figure 4: policy and cost function for exact policy iteration

From the simulation results, it is observed that the approximate policy iteration can lead to a near optimal policy with only nine features, linear approximation architecture and batch mode least squares solver. Although it is slower than exact policy iteration in convergence rate, it requires much less storage, and does not need to solve a large linear equation, so it has the potential to solve large problems.

This case study also provides the starting point for further work on approximate policy iteration.

Possible future research topics include: 1) trying nonlinear approximate architectures for the cost-to-go approximation; 2) trying incremental gradient methods for the least squares solver; 3) trying other forms of features; 4) considering more inventory levels; 5) considering the situation when machines can be purchased, discarded, or sent to reserve, and maintenance is required.

Acknowledgement:

This work was supported in part by the National Science Foundation under Grant DMI-9713720, in part by the Semiconductor Research Corporation under Grant 97-FJ-491, and in part by a fellowship from General Electric Corporate Research and Development through the Institute for Systems Research.

References

- [1] S. Bhatnagar, E. Fernandez-Gaucherand, M. C. Fu, Y. He, and S. I. Marcus, "A Markov decision process capacity expansion and allocation," in *Proc. of the 38th Conference on Decision and Control*, Phoenix, Arizona, 1999, pp. 1156–1161.
- [2] S. Bhatnagar, M. C. Fu, S. I. Marcus, and Y. He, "Markov decision processes for semiconductor fab-level decision making," in *Proc. of the IFAC 14th Triennial World Congress*, Beijing, P. R. China, 1999, pp. 145–150.
- [3] M. L. Puterman, *Markov Decision Processes*, John Wiley & Sons, Inc., New York, 1994.
- [4] D. P. Bertsekas, *Dynamic Programming and Optimal Control Vol 1 & 2*, Athena Scientific, Belmont, Massachusetts, 1995.
- [5] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, Belmont, Massachusetts, 1996.
- [6] A. M. Law and W. D. Kelton, *Simulation Modeling & Analysis*, McGraw - Hill, Inc., New York, 1991.