Homeworks Business Analytics
Homework2:
Politica di controllo della produzione
con magazzino a capacità finita e domanda stocastica

Enrico Agrippino (291272), Teresa Di Renzo (290523)

Luglio 2021

Sommario

Obiettivo di questo homework è trovare la politica ottimale per la gestione di una linea produttiva e del relativo magazzino. Per ogni time bucket, la macchina può produrre una quantità fissa di uno solo fra i prodotti A o B, oppure restare in stato idle. Quando il prodotto realizzato è diverso dal precedente, si introduce un costo e un tempo di setup. La domanda è supposta stocastica: sequenza di variabili aleatorie i.i.d. di cui conosciamo la distribuzione.

Per trovare la policy ottimale faremo uso della programmazione dinamica: usando un approccio backward, per ogni istante di tempo andremo a calcolare il valore della Value function per ogni possibile valore della variabile di stato, scegliendo ogni volta l'azione migliore. Tale algoritmo è implementato nella funzione *MakePolicy*. Otterremo così *valueTable* e *actionTable*: due tabelle contenenti i valori della Value function e le decisioni della politica ottima per ogni istante di tempo e per ogni valore della variabile di stato.

Fissato uno stato iniziale, andremo poi a testare il valore della politica ottima trovata tramite simulazione Monte Carlo out of sample (funzione *SimulatePolicy*), confrontando tale risultato con quello contenuto in *valueTable*. I due risultati coincidono (al netto di piccole oscillazioni dovute alla stocasticità del metodo Monte Carlo) per ogni valore iniziale.

I codici MATLAB *SimulatePolicy* e *MakePolicy* sono riportati per completezza espositiva anche in questo report, oltre che nella cartella condivisa.

Andremo poi a verificare l'impatto che hanno i parametri del problema sulla soluzione finale sia in termini di politica ottima che di value function.

Nell'ultima sezione spiegheremo come si potrebbero togliere le due semplificazioni aggiunte alla richiesta originaria: quattro prodotti anziché due, e time bucket non coincidente con l'unità di avanzamento del tempo.

Indice

1 Modello e semplificazioni adottate

L'algoritmo implementato per ottenere la politica ottimale corrisponde alla soluzione del modello dato dalle assunzioni seguenti.

- La linea produttiva (equivalente a una macchina) può produrre due prodotti: A e B.
- Il processo produttivo è suddiviso temporalmente in time bucket fissi: una volta deciso quale prodotto produrre, la produzione andrà avanti fino alla fine del time bucket. Di conseguenza, la scansione del tempo nell'algoritmo di programmazione dinamica sarà data dal susseguirsi dei time bucket.
- Le quantità che la macchina è in grado di produrre per time bucket è fissa ed è data dagli interi qA e qB. La produzione si conclude complessivamente alla fine di ogni time bucket.
- La capacità del magazzino ($maxOnHand$) è limitata. In particolare stiamo supponendo due magazzini distinti per i due prodotti con uguale capacità (quindi se abbiamo già in magazzino la quantità massima stoccabile di A non potremo produrre A, indipendentemente dal valore del magazzino di B).
- I costi di produzione unitari sono in generale diversi per i due prodotti: tali costi sono dati dai numeri reali positivi cA e cB.
- La macchina può anche rimanere in stato idle, non producendo nulla.
- Quando decidiamo di produrre un prodotto diverso da quello prodotto precedentemente (indipendentemente dal fatto se al tempo precedente la macchina si trovasse in stato idle o meno), abbiamo un costo e un tempo di setup. Il costo di setup è fisso ed è dato dal numero reale positivo w; il tempo di setup è supposto sempre inferiore a un time bucket e si traduce dunque in una minore quantità di prodotto realizzato, uguale per i due prodotti: l'intero s. Ad esempio, se l'ultimo prodotto è stato il prodotto A e nel time bucket attuale decidiamo di produrre il prodotto B, la quantità che saremo in grado di produrre è data da qB - s.
- Il costo unitario di giacenza a magazzino per time bucket è dato da h ed è uguale per i due prodotti.
- Nel caso in cui non riuscissimo a far fronte alla domanda con i prodotti presenti in magazzino (stockout), avremmo una penalità per la domanda insoddisfatta. Tale penalità, uguale per i due prodotti, è lineare e data dal coefficiente b. La domanda persa non è più recuperabile negli istanti successivi.
- L'orizzonte temporale è finito ed è dato da $horizon$. Come analizzeremo in seguito, l'algoritmo riesce a trovare una policy stazionaria al tempo iniziale per valori di $horizon$ non troppo piccoli.
- Supponiamo di osservare all'inizio di un time bucket la domanda accumulatasi nel time bucket precedente. Di conseguenza vendiamo solo in istanti di tempo discreti dati dal susseguirsi dei time bucket.
- La domanda è supposta discreta e finita: si tratta dunque di una distribuzione multinomiale. La funzione di probabilità della domanda, che stiamo supponendo di conoscere, è in generale diversa per i due prodotti.
- Al tempo finale supponiamo di riuscire a vendere tutti i prodotti in magazzino esattamente al prezzo di costo.

2 MakePolicy

Vediamo adesso i punti principali dell'algoritmo implementato per l'apprendimento della policy. L'approccio usato è quello della programmazione dinamica a tempo finito, backward nel tempo. Stiamo supponendo di essere nel discreto finito piccolo, di conseguenza possiamo permetterci una rappresentazione tabellare per ogni valore della variabile di stato e del tempo.

Definiamo la variabile di stato come un vettore (nel seguito chiamato **S**) di tre elementi:

1. il livello di magazzino per il prodotto A: *onHandA*,
2. il livello di magazzino per il prodotto B: *onHandB*,
3. l'ultimo prodotto realizzato dalla linea produttiva: tale informazione è salvata in *lastP*, che può assumere valore 1 (ultimo prodotto = A) o 2 (ultimo prodotto = B).

La terza componente è resa necessaria dalla presenza del setup: la value function e la policy dipenderanno non solo dal livello di magazzino per i due prodotti, ma anche dall'ultimo prodotto realizzato.

Ora che abbiamo definito le variabili di stato, possiamo finalmente definire la value function. Andremo a considerare esclusivamente i costi: il problema di ottimizzazione che risolviamo ad ogni istante di tempo per decidere l'azione migliore sarà dunque un problema di minimo. L'incentivo alla produzione, non essendo presente il profitto nel nostro modello, è dunque dato esclusivamente dal coefficiente b che penalizza la domanda inevasa. Ad ogni istante dobbiamo prendere la decisione x, che può assumere valori $x = 0$ (nel prossimo time bucket non produciamo nulla), $x = 1$ (produciamo A) o $x = 2$ (produciamo B). La value function all'istante t in funzione del vettore delle variabili di stato $\mathbf{S}$ è:

$$V_t(\mathbf{S}) = \min_{x \in \mathcal{X}} \left\{ C(\mathbf{S}, x) + \gamma \mathbb{E}\left[H(\mathbf{S}, x, d_{t+1}) + B(\mathbf{S}, x, d_{t+1}) + V_{t+1}(\mathbf{S}, x, d_{t+1}) \right] \right\} \tag{1}$$

dove:

- $\mathcal{X}$ è l'insieme delle azioni ammissibili per l'attuale valore della variabile di stato (ricordiamo che il magazzino ha capacità finita).
- $C(\mathbf{S}, x)$ è la funzione dei costi di produzione che pago all'istante attuale t: tali costi dipendono, oltre che dalla decisione presa x, anche da *lastP*, perchè devo essere in grado di capire se abbiamo i costi e i tempi di setup.
- $H(\mathbf{S}, x, d_{t+1})$ è la funzione dei costi di magazzino che pagherò all'istante successivo. Tali costi dipendono, oltre che dagli attuali livelli di magazzino e dalle quantità prodotte in base a x e a *lastP*, anche dalla domanda d_{t+1} che osserverò solo alla fine del prossimo time bucket.
- $B(\mathbf{S}, x, d_{t+1})$ è la funzione dei costi per domanda inevasa in caso di stockout che pagherò all'istante successivo. Valgono le stesse considerazioni di $H(\mathbf{S}, x, d_{t+1})$.
- γ è il fattore di sconto.

Per risolvere l'equazione ricorsiva definita in (1), occorre fornire un valore per la value function al tempo finale $T = \textit{horizon}$. Abbiamo posto per semplicità $V_T(\mathbf{S}) = 0 \quad \forall \mathbf{S}$, ipotizzando che al tempo finale l'azienda sia in grado di vendere tutti i prodotti presenti in magazzino ad un prezzo esattamente pari ai costi.

Resta da analizzare come gestire il valore atteso presente nell'equazione (1). Dato che lo spazio degli stati, e con esso la domanda, è discreto, finito e relativamente piccolo, e dato che conosciamo la distribuzione di probabilità della domanda, possiamo calcolare il valore atteso semplicemente facendo una somma pesata per le probabilità di ogni possibile valore di domanda. In MATLAB abbiamo quindi creato il vettore *nextInv* che contiene tutti i livelli possibili di magazzino dati i livelli attuali e i valori di domanda futuri. *nextInv* avrà dunque dimensioni $2 \times \textit{maxDemand}$, dove *maxDemand* è il valore massimo di domanda, uguale per i due prodotti (ma ciò non vuol dire che debba esserlo anche nel modello, basta aggiungere delle probabilità nulle per il prodotto che ha valore massimo di domanda inferiore: è solo una necessità implementativa dell'algoritmo).

2.1 Codice MATLAB

Riportiamo adesso il codice MATLAB commentato. Dopo le inizializzazioni iniziali, facciamo partire i cicli for sul tempo (backward) e sulle tre variabili di stato. All'interno del ciclo più interno andiamo a provare tutte e tre le decisioni possibili e al termine scegliamo l'azione che ha ottenuto un valore della value function minore, andando a salvare la decisione e la value function così ottenute.

```
function [valueTable,actionTable] = MakePolicy(maxOnHand, qA, qB, cA, cB,...
    h, b, s, w, horizon, demandProbs, gamma)
% maxOnHand livello max magazzino per prodotto
% qA, qB quantità dei prodotti A e B prodotte in un time bucket
% cA, cB costi produzione A e B
% h costo magazzino
% b penalità domanda insoddisfatta
% s quantità prodotta in meno (se setup)
% w costo di setup
% horizon orizzonte temporale
% demandProbs valori di prob discreti della domanda

%x=0,1,2 decisione se produrre A (1), B (2), non produrre (0)

% valueTable conterrà i valori dello stato in un determinato tempo:
% è un array 4D, dove le prime tre componenti sono le variabili
% di stato, l'ultima componente è il tempo
valueTable = zeros(maxOnHand+1,maxOnHand+1,2,horizon+1);
% actionTable conterrà la best policy per ogni stato in un determinato tempo
actionTable = zeros(maxOnHand+1,maxOnHand+1,2,horizon);
% maxDemand è la domanda massima possibile per ogni istante di tempo,
% corrispondente alla domanda massima fra i due prodotti (demandProbs è una
% matrice maxDemand X 2)
maxDemand = length(demandProbs(:,1))-1;
demandValues = [0:maxDemand;0:maxDemand];
% valori di domanda per il prodotto A (prima riga) e B (seconda riga)
% le probabilità corrispondenti si hanno in demandProbs

% ciclo backward sul tempo
for t=(horizon-1):-1:0
    % 3 for sulle variabili di stato:
    for onHandA=0:maxOnHand
        for onHandB=0:maxOnHand
            for lastP=1:2

                % inizializziamo minCost e best_X, corrispondenti ai valori
                % da inserire rispettivamente nella valueTable e
                % actionTable
                minCost=inf;
                best_x=0;
                % valutiamo tutte le decisioni possibili:
                x=0; % prima decisione: non produciamo
                nextInv=[onHandA;onHandB]-demandValues; % nextInv
                % sono i possibili valori di magazzino (o di
                % backlog) dopo aver osservato la domanda;
                % calcoliamo adesso al calcolo della valueFunction
                % nel caso in cui la decisione x=0
                expCost = gamma*(dot(demandProbs(:,1),h*max(0,(nextInv(1,:))))+...
                    +dot(demandProbs(:,2),h*max(0,(nextInv(2,:))))+...
                    +dot(demandProbs(:,1),b*max(0,(-nextInv(1,:))))+...
                    +dot(demandProbs(:,2),b*max(0,(-nextInv(2,:)))))+...
                    gamma*demandProbs(:,1)'*squeeze(valueTable(max(nextInv(1,:),0)+1,...
                    max(nextInv(2,:),0)+1,lastP,t+2))*demandProbs(:,2);
                % controlliamo se siamo al minimo (in questo caso
                % lo sarà sempre perchè è la prima decisione che
                % testiamo)
```

```
                if expCost < minCost
                    minCost=expCost;
                    best_x=x;
                end

                x=1; % seconda decisione: produciamo A
                flag_setup=(x~=lastP); % 1 in caso di setup 0 altrimenti
                if onHandA+qA-flag_setup*s<=maxOnHand % controlliamo se
                    % è possibile produrre A (cioè se c'è ancora spazio nel magazzino di A)
                    q=[qA-flag_setup*s;0]; % andiamo a sottrarre a qA
                    % la quantità che non produciamo in caso di setup
                    nextInv=[onHandA;onHandB]+q-demandValues;
                    expCost = cA*q(1) + flag_setup*w +...
                        +gamma*(dot(demandProbs(:,1),h*max(0,(nextInv(1,:))))+...
                        +dot(demandProbs(:,2),h*max(0,(nextInv(2,:))))+...
                        +dot(demandProbs(:,1),b*max(0,(-nextInv(1,:))))+...
                        +dot(demandProbs(:,2),b*max(0,(-nextInv(2,:)))))+...
                        +gamma*demandProbs(:,1)'*squeeze(valueTable(max(nextInv(1,:),0)+1,...
                        max(nextInv(2,:),0)+1,x,t+2))*demandProbs(:,2);

                    if expCost < minCost
                        minCost=expCost;
                        best_x=x;
                    end
                end

                x=2; % terza decisione: produciamo B
                flag_setup=(x~=lastP);
                if onHandB+qB-flag_setup*s<=maxOnHand
                    q=[0;qB-flag_setup*s];
                    nextInv=[onHandA;onHandB]+q-demandValues;
                    expCost = cB*q(2) + flag_setup*w +...
                        gamma*(dot(demandProbs(:,1),h*max(0,(nextInv(1,:))))+...
                        +dot(demandProbs(:,2),h*max(0,(nextInv(2,:))))+...
                        +dot(demandProbs(:,1),b*max(0,(-nextInv(1,:))))+...
                        +dot(demandProbs(:,2),b*max(0,(-nextInv(2,:)))))+...
                        gamma*demandProbs(:,1)'*squeeze(valueTable(max(nextInv(1,:),0)+1,...
                        max(nextInv(2,:),0)+1,x,t+2))*demandProbs(:,2);

                    if expCost < minCost
                        minCost=expCost;
                        best_x=x;
                    end
                end

                % aggiorniamo adesso la value e action table
                valueTable(onHandA+1,onHandB+1,lastP, t+1)=minCost;
                actionTable(onHandA+1,onHandB+1,lastP,t+1)=best_x;



            end %lastP
        end % onHandB
    end % onHandA
end %tempi
end
```

3 SimulatePolicy

Una volta trovati i valori della Value function (salvati in *valueTable*) e la corrispondente politica ottima (contenuta in *actionTable*), andiamo a testare il valore della politica ottima con simulazioni Monte Carlo. Ciò che vogliamo verificare è che il costo medio trovato dalla funzione *SimulatePolicy* per un dato stato iniziale $\mathbf{S}_0$ sia equivalente al valore in *valueTable* all'istante $t = 0$ per lo stato corrispondente a $\mathbf{S}_0$. La funzione *SimulatePolicy* richiede dunque in input, oltre ai dati del problema, anche la *actionTable* trovata in precedenza, il numero di scenari Monte Carlo su cui testarla e lo stato iniziale $\mathbf{S}_0$ (che nel codice MATLAB corrisponde ai valori `startState` e `lastP`). In output restituirà, per ogni scenario, la value function al tempo iniziale per lo stato iniziale passato in input ($V_0(\mathbf{S}_0)$) ottenuta seguendo la politica.

In questo caso la simulazione è effettuata procedendo in avanti nel tempo. Per ogni istante di tempo simuliamo la domanda (essendo la domanda una sequenza di variabili aleatorie i.i.d., ciò è equivalente a simulare un sample path intero per ogni scenario, ma permette di risparmiare allocazione di memoria) che andremo a usare per determinare il valore di *nextInv* e con esso i costi. Al tempo finale otterremo così il valore della value function per lo scenario corrente, dopodichè procederemo allo stesso modo per lo scenario successivo.

3.1 Codice MATLAB

Riportiamo adesso il codice MATLAB commentato.

```
function valueFunction = SimulatePolicy(actionTable, demandProbs, maxOnHand, ...
    cA, cB, qA, qB, h, b, s, w, horizon, numScenarios, startState, lastP_0, gamma)
% Creiamo innanzitutto le distribuzioni di probabilità multinomiale sulla
% base delle probabilità date nell'input demandProbs:
pdA = makedist('Multinomial','probabilities',demandProbs(:,1));
pdB = makedist('Multinomial','probabilities',demandProbs(:,2));
% inizializzazione del vettore delle value function per
% lo stato iniziale (startState, lastP_0) al tempo iniziale di dimensione
% pari al numero di scenari:
valueFunction = zeros(numScenarios,1);
for k = 1:numScenarios % ciclo sugli scenari
    state = startState; % inizializzazione stato iniziale
    lastP = lastP_0; % inizializzazione ultimo prodotto
    expCost = 0; % inizializzazione valore da inserire nella value function

    % cicliamo in avanti sui tempi
    for t = 1:horizon
        % simuliamo le domande per i due prodotti, dato che le probabilità
        % sono iid, simuliamo due scalari a ogni scenario e a ogni tempo
        % (cosi' risparmiamo memoria)
        dA=random(pdA)-1;
        dB=random(pdB)-1;
        % creiamo il vettore della domanda attuale
        domanda=[dA;dB];
        % leggiamo l'azione dettata dalla policy nello stato attuale
        x = actionTable(state(1)+1,state(2)+1,lastP,t);
        flag_setup=(x~=lastP); % 1 in caso di setup 0 altrimenti
        % in base all'azione cambia il contributo alla value function:
        if x==0
            nextInv=state-domanda; % nextInv ci dà lo stato attuale
            % dopo aver osservato la domanda
            expCost= gamma*(expCost+h*max(0,nextInv(1)) + h*max(0,nextInv(2))...
                +b*max(0,-nextInv(1)) + b*max(0,-nextInv(2)));
        else if x==1 && ((state(1)+qA-flag_setup*s)<=maxOnHand)
                q=[qA-flag_setup*s;0];
                nextInv=state+q-domanda;
```

```
                % nextInv ci dà lo stato attuale dopo aver osservato
                % la domanda e aver prodotto A
                expCost= gamma*expCost + cA*q(1) + gamma*(h*max(0,nextInv(1))...
                    +h*max(0,nextInv(2)) + b*max(0,-nextInv(1)) +...
                    b*max(0,-nextInv(2))) + flag_setup*w;
                lastP=1; % aggiorniamo l'ultimo prodotto
            else if x==2 && (state(2)+qB<=maxOnHand)
                    flag_setup=(x~=lastP);
                    q=[0;qB-flag_setup*s];
                    nextInv=state+q-domanda;
                    expCost= gamma*expCost+ cB*q(2) + gamma*(h*max(0,nextInv(1))...
                        + h*max(0,nextInv(2)) + b*max(0,-nextInv(1))...
                        + b*max(0,-nextInv(2))) + flag_setup*w;
                    lastP=2; % aggiorniamo l'ultimo prodotto
                end
            end
        end % fine if

        state = max(0, nextInv); % aggiorniamo lo stato
    end % fine ciclo sui tempi

    % aggiorniamo la value function per lo scenario k-esimo
    valueFunction(k) = expCost;

end % fine ciclo sugli scenari
end %function
```

4 Analisi risultati

In questa sezione analizziamo i risultati ottenuti con il metodo implementato, studiando in particolare l'impatto di diversi fattori sul valore della *valueTable* all'istante iniziale e sull'*actionTable*, dati *startState* e *lastP*.

A tal fine, settiamo innanzitutto dei parametri di default da passare alla funzione *MakePolicy*; successivamente variamo 1 o 2 parametri alla volta per studiarne l'impatto.

Riportiamo di seguito i parametri di default scelti:

- $maxOnHand = 15$;
- $qA = 4$;
- $qB = 2$;
- $cA = 0.4$;
- $cB = 0.35$;
- $h = 0.15$;
- $b = 0.7$;
- $s = 1$;
- $w = 0.8$;
- $horizon = 20$;
- $gamma = 0.99$;

- $demandProbs = \begin{bmatrix} 0.1 & 0.2 \\ 0.2 & 0.3 \\ 0.3 & 0.4 \\ 0.3 & 0.1 \\ 0.1 & 0 \end{bmatrix}$;

- $startState = [0, 0]$;

- $lastP = 1$.

Con questi parametri di default osserviamo un valore della *valueTable* all'istante iniziale pari a 38.2967. Riportiamo in figura 1 l'*actionTable* al tempo iniziale $t = 0$ (ma in MATLAB questo corrisponde all'indice 1) e per lastP= (ultimo prodotto A), e in figura 2 l'*actionTable* al tempo iniziale e per lastP= 2 (ultimo prodotto B); sulle righe varia la quantità di magazzino per il prodotto A, mentre sulle colonne la quantità di magazzino del prodotto B.

```
val(:,:,1,1) =

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Figura 1: *actionTable* per $t = 1$ e lastP= 1.

```
val(:,:,2,1) =

   2   2   1   1   1   1   1   1   1   1   1   1   1   1   1   1
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Figura 2: *actionTable* per $t = 1$ e lastP= 2.

Possiamo osservare come nel caso in cui *lastP*= 1, produciamo il prodotto A se ne abbiamo poco in magazzino, mentre produciamo B se, oltre ad avere poche unità di B in magazzino, abbiamo elevate quantità di A. Infatti possiamo notare come a parità di livello di magazzino per il prodotto A, produciamo A a prescindere dal livello di magazzino di B; viceversa, a parità di livello di magazzino di B, questo viene prodotto solo per un elevato magazzino di A.
Nel caso in cui *lastP*= 2, invece, viene prodotto B per bassi valori di magazzino di B a prescindere dal livello di magazzino di A.
Analizzando l'*actionTable* per gli istanti successivi, osserviamo come progressivamente aumentino i livelli di magazzino per cui non produciamo alcun prodotto: questo fatto può essere spiegato ricordando che abbiamo imposto il valore della value function al tempo finale pari a 0.

Procediamo quindi con la variazione di un parametro, imponendo *maxOnHand*= 30. Ai nuovi dati associamo una differente probabilità di domanda:

$$demandProbs = \begin{bmatrix} \texttt{repmat}(0,8,1) & 0 & 0 & 0.1 & 0.2 & 0.3 & 0.3 & 0.1 \\ \texttt{repmat}(0,8,1) & 0.2 & 0.3 & 0.4 & 0.1 & 0 & 0 & 0 \end{bmatrix}.$$

Si ottiene come valore della *valueTable* all'istante iniziale 249.9683: peggiora sensibilmente rispetto a quella ottenuta con i parametri di default. Infatti, aumentando la capacità massima del magazzino e la domanda aumentano i costi di giacenza.
Osservando l'*actionTable* al tempo iniziale nel caso *lastP*= 1, notiamo come la politica suggerisca di produrre sempre A finché non si raggiunge un livello di magazzino tale per cui la sua produzione sforerebbe *maxOnHand*. Nel caso *lastP*= 2, è suggerita la produzione sia di A che di B a seconda dei diversi livelli di magazzino. A prescindere da *lastP* quindi sono più i livelli di magazzino per cui è conveniente produrre A. Per quanto riguarda invece i tempi finali, se *lastProd* = 1 non produciamo niente, mentre se *lastProd* = 2 vi sono alcuni livelli di magazzino per cui è consigliato produrre B.

Lasciando *maxOnHand* = 30, cambiamo ora *demandProbs* per avere la domanda più variabile:

$$demandProbs = \begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \end{bmatrix}.$$

In questo modo, la probabilità è la stessa per ogni possibile valore di domanda (interi da 0 a 9). Si ottiene un valore di *valueTable* all'istante iniziale pari a 100.1359. Possiamo quindi osservare un netto miglioramento. Ciò può essere dovuto al fatto che stiamo riducendo la media della domanda, sebbene ne stiamo aumentando la variabulità.

Riportando quindi il parametro *maxOnHand* al valore di default, ovvero 15, e lasciando invariato *demandProbs*, il valore della *valueTable* che si ottiene è ancora 100.1359: questo ci fa supporre che la probabilità di domanda abbia un impatto maggiore rispetto alla capacità di magazzino.

Consideriamo ora una domanda meno variabile:

$$demandProbs = \begin{bmatrix} 0 & 0 & 0 & 0.1 & 0.8 & 0.1 \\ 0 & 0.1 & 0.8 & 0.1 & 0 & 0 \end{bmatrix}.$$

Si ottiene come valore della *valueTable* all'istante iniziale 56.0827: questo miglioramento è dovuto al fatto che la massa di probabilità è concentrata per poche date domande per prodotto, quindi si può predire la domanda con più precisione.
Osservando l'*actionTable* al tempo $t = 1$ nel caso *lastP*= 1, notiamo che produciamo A nel caso il suo livello di magazzino sia basso, mentre produciamo B se il livello di magazzino di B è basso e il livello di magazzino di A è elevato. In questo caso, infatti, si sta pagando un alto prezzo di giacenza, e per ridurre i costi si può mirare a diminuire il costo della domanda insoddisfatta producendo B.
Nel caso in cui *lastP*= 2, osserviamo che l'*actionTable* suggerisce di produrre B per più livelli di magazzino, ma comunque non per livelli di magazzino di B più alti di 5. Infatti considerando la domanda che abbiamo imposto, con un livello di magazzino pari a 5 la probabilità di soddisfare due domande è molto elevata: producendo B si aggiungerebbero costi di giacenza senza ridurre alcun costo di domanda insoddisfatta.

Riportiamo ora *demandProbs* al parametro di default e variamo il rapporto tra costo di magazzino e costo di setup.
Imponendo $h = 0.2$ e $w = 0.3$, otteniamo un valore della *valueTable* pari a 38.3373, quindi assistiamo a un miglioramento rispetto ai parametri di default. Confrontando le *actionTable* notiamo come produciamo B per più livelli di magazzino rispetto a prima.

Diminuiamo ora il rapporto, imponendo $h = 0.1$ e $w = 1.2$. Il valore della *valueTable* che si ottiene è pari a 37.0361, si assiste pertanto a un miglioramento: infatti, diminuendo il costo di giacenza i costi totali si riducono. Dall'*actionTable* osserviamo che nel caso *lastP*= 1 produciamo quasi sempre A, mentre nel caso *lastP*= 2 produciamo quasi sempre B: aumentando i costi di setup non è infatti conveniente cambiare prodotto in produzione.

Possiamo concludere che l'impatto del rapporto tra costo di magazzino e costo di setup è consistente.

Studiamo ora l'impatto del costo della domanda insoddisfatta. Definendo $b = 1.5$, il valore della *valueTable* peggiora: otteniamo infatti 54.4677. Osservando l'*actionTable* possiamo notare come siano molto più numerosi i valori di magazzino per cui produciamo qualcosa; inoltre accade più spesso di produrre prodotti diversi da *lastP*. Possiamo quindi ipotizzare che il valore della *valueTable* aumenti perché un costo della domanda insoddisfatta maggiore conduce a produrre di più, aumentando il costo di giacenza e il costo di setup, oltre ovviamente all'effetto del coefficiente b.

Per valutare l'effetto della capacità produttiva, ripristiniamo il valore del costo della domanda insoddisfatta di default, e imponiamo $qA = 8, qB = 6$. Otteniamo un valore della *valueTable* all'istante iniziale pari a 42.6181, risultato peggiore rispetto a quello ottenuto con i parametri iniziali.
Osservando l'*actionTable* possiamo notare come la politica suggerisca in generale di produrre meno: questo perché produrre ha un costo maggiore e porta ad un aumento del costo di giacenza; inoltre sono più numerosi i livelli di magazzino per cui producendo si sforerebbe *maxOnHand*.

Diminuendo il costo di giacenza unitario, $h = 0.05$, la politica suggerisce di produrre maggiormente, ma non in modo significativo. Anche aumentando il costo per la domanda insoddisfatta, $b = 1.5$, osserviamo come nell'*actionTable* le decisioni di produrre siano per pochi livelli di magazzino in più.

Possiamo spiegare questo comportamento notando che la distribuzione di probabilità della domanda resta inalterata: si producono quindi quantità maggiori senza che la richiesta sia aumentata.

Effettuati questi test, possiamo concludere che la capacità produttiva abbia grande impatto.

Un fattore che sembra non avere grande impatto sul valore della *valueTable* all'istante iniziale è invece la quantità s che non viene prodotta in caso di setup: imponendo $s = 0$ si ottiene lo stesso valore ottenuto con i parametri di default.

Studiato l'effetto di questi fattori, aumentiamo ora l'orizzonte temporale. Come atteso, il valore della *valueTable* all'istante iniziale è maggiore perché è necessario sommare un numero di costi più elevato. È interessante notare come invece l'*actionTable* per i tempi iniziali sia la medesima: questo significa che per tempi sufficientemente lontani dall'orizzonte temporale, la politica trovata è stazionaria.

Dati i parametri di default considerati, possiamo concludere che è consigliabile per più livelli di magazzino e per più tempi produrre A: questo perché ha costo unitario minore e valore di quantità prodotte per time bucket maggiore, per cui diminuisce il costo di domanda insoddisfatta e la diminuzione della quantità prodotta a causa del setup ha effetto minore.

5 Modifiche al modello

Riportiamo in questa sezione due modifiche al modello su cui abbiamo lavorato e che abbiamo provato a implementare, ottenendo però risultati insoddisfacenti. La prima modifica riguarda il numero di prodotti, 4 anziché 2. La seconda la scansione del tempo: abbiamo definito come unità di tempo un tempo differente dal time bucket (il time bucket e il setup possono durare più unita di tempo), per cui è stato possibile definire un tempo di setup anche maggiore del time bucket.
Analizziamo ora la prima modifica.

5.1 4 prodotti

Nell'estensione a una linea produttiva capace di produrre 4 prodotti anziché 2, si è utilizzato il modello esposto nella sezione introduttiva (1), con le seguenti modifiche:

- La linea produttiva può produrre 4 prodotti: A1, A2, B1 e B2. I prodotti sono suddivisi in due famiglie (A e B), pertanto A1 e A2 appartengono alla stessa famiglia, mentre B1 appartiene alla stessa famiglia di B2.
- La quantità che la macchina può produrre per time bucket è fissa e dipende dal prodotto in produzione; tale quantità è data dagli interi $qA1, qA2, qB1, qB2$.

- I costi di produzione sono differenziati a seconda del prodotto e sono dati dai numeri reali positivi $cA1, cA2, cB1, cB2$.
- Nel caso si produca un prodotto diverso dall'ultimo prodotto dalla macchina, si ha un costo e un tempo di setup che varia a seconda che il nuovo prodotto sia della stessa famiglia dell'ultimo o meno. Avremo quindi due costi di setup $w1$ e $w2$: il primo nel caso il prodotto precedente e il nuovo appartengano alla stessa famiglia, il secondo se i due prodotti sono di famiglie differenti. Il tempo di setup è supposto inferiore a un time bucket e si traduce in una minore quantità di prodotto realizzato, come nel modello del paragrafo 1. Tale quantità è pari a $s1$ se la famiglia dei due prodotti è la medesima, altrimenti è pari a $s2$.

L'algoritmo implementato in MATLAB per l'apprendimento della policy con questo modello è riportato nella cartella condivisa con il nome *MakePolicy4_prova.m*.
Riportiamo di seguito le differenze rispetto all'algoritmo illustrato nel paragrafo 2.
In questo caso la variabile di stato $\mathbf{S}$ è un vettore di 5 elementi:

1. il livello di magazzino per il prodotto A1: *onHandA1*;
2. il livello di magazzino per il prodotto A2: *onHandA2*;
3. il livello di magazzino per il prodotto B1: *onHandB1*;
4. il livello di magazzino per il prodotto B2: *onHandB2*;
5. l'ultimo prodotto realizzato dalla linea produttiva: tale informazione è salvata in *lastP*, che può assumere valore 1 (ultimo prodotto A1), 2 (ultimo prodotto A2), 3 (ultimo prodotto B1), 4 (ultimo prodotto B2).

Come per il modello precedente, la value function all'istante t in funzione del vettore delle variabili di stato $\mathbf{S}$ è data da:

$$V_t(\mathbf{S}) = \min_{x \in \mathcal{X}} \{C(\mathbf{S}, x) + \gamma \mathbb{E}\left[H(\mathbf{S}, x, d_{t+1}) + B(\mathbf{S}, x, d_{t+1}) + V_{t+1}(\mathbf{S}, x, d_{t+1})\right]\}.$$

Ad ogni istante di tempo bisogna prendere la decisione x, che in questo caso assumerà valore 0 (non produciamo alcun prodotto), 1 (produciamo A1), 2 (produciamo A2), 3 (produciamo B1), 4 (produciamo B2).

L'algoritmo così implementato si è rivelato non essere efficiente: mantenendo gli stessi valori di *maxOnHand* e *horizon* utilizzati per il modello precedente, il tempo di calcolo è risultato essere eccessivamente elevato.
Inoltre, vi è un'incongruenza tra i valori della value function salvati in *valueTable* ed i valori ottenuti con simulazioni Monte Carlo.

5.2 Modifica alla scansione del tempo

Vediamo ora la seconda possibile modifica al modello illustrato al paragrafo 1: la scansione del tempo. A differenza del modello precedente in cui l'unità di misura di avanzamento del tempo coincideva con il time bucket, in questa modifica abbiamo utilizzato una differente unità di misura: in questo modo è stato possibile definire a piacere sia il time bucket che il tempo di setup, ed in particolare è stato possibile imporre un tempo di setup maggiore del time bucket.
Per questo modello, infatti, s non indica più la quantità di prodotto non realizzata a causa del setup, ma indica effettivamente un tempo. Così facendo, abbiamo supposto che in caso di setup la macchina produca la stessa quantità prodotta in assenza di setup, ma in un tempo pari a $t_B + s$ anziché t_B (dove t_B indica il time bucket).

L'algoritmo implementato in MATLAB per l'apprendimento della policy con questo modello è riportato nella cartella condivisa con il nome *MakePolicy_setup_maggiore_tB_prova.m*. Illustriamo di seguito le differenze rispetto all'algoritmo descritto al paragrafo 2.

In questo caso la variabile di stato $\mathbf{S}$ è un vettore di 4 elementi:

1. il livello di magazzino per il prodotto A: *onHandA1*;

2. il livello di magazzino per il prodotto B: *onHandA2*;

3. l'ultimo prodotto realizzato dalla linea produttiva: tale informazione è salvata in *lastP*, che può assumere valore 1 (ultimo prodotto A), 2 (ultimo prodotto B).

4. L'informazione salvata in *avail* riguardante la disponibilità della macchina al tempo t: *avail* può assumere valore 1, nel caso la macchina sia disponibile, e tutti i valori compresi tra 2 e $t_B + s + 1$, nel caso la macchina sia impegnata in un ciclo di produzione. Per il valore *avail*$= t_B + s + 1$ la macchina ha appena iniziato il ciclo di produzione con setup, per cui è necessario attendere il time bucket ed il tempo di setup prima di poter prendere una decisione; al diminuire dei valori di *avail* il tempo di attesa diminuisce, fino a *avail*$= 2$, ultimo istante per cui la macchina è occupata. Nel caso la produzione non richieda setup, ad inizio ciclo di produzione si avrà invece *avail*$= t_B + 1$.

Come per entrambi i modelli già visti, la value function all'istante t in funzione del vettore delle variabili di stato $\mathbf{S}$ è data da:

$$V_t(\mathbf{S}) = \min_{x \in \mathcal{X}} \{C(\mathbf{S}, x) + \gamma \mathbb{E}\left[H(\mathbf{S}, x, d_{t+1}) + B(\mathbf{S}, x, d_{t+1}) + V_{t+1}(\mathbf{S}, x, d_{t+1})\right]\}.$$

Per questo modello, la decisione x da prendere ad ogni istante di tempo può assumere valore 0 (non produciamo), 1 (produciamo A), 2 (produciamo B).
È importante notare come in realtà sia possibile prendere una decisione solo se la macchina è disponibile: nel caso non lo sia, la decisione di non produrre niente e aspettare è obbligata. Tuttavia, ad ogni istante di tempo si osserva una domanda, per cui aggiorniamo *nextInv* (presentato alla sezione 2) ad ogni tempo.

Per quanto riguarda il codice MATLAB di questo modello, questo rispetto a quello illustrato al paragrafo 2.1 presenta un ulteriore ciclo sulla quarta variabile di stato *avail*: nel caso *avail* $= 1$ si procede come per il modello precedente, con l'eccezione che nel caso la decisione sia di produrre, *expCost* terrà conto delle value function non solo dell'istante successivo ma di tutti i tempi necessari affinché la macchina sia nuovamente disponibile; nel caso *avail* $\neq 1$ si calcola *nextInv* senza produrre nulla e si scala il tempo di attesa per poter prendere una nuova decisione.

L'algoritmo così implementato non ha portato a risultati plausibili per quanto riguarda l'*actionTable*: la politica propone la produzione solo per istanti prossimi a *horizon*, mentre per tutto il restante arco temporale suggerisce di non produrre.