Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра математики факультету інформатики

A multicriteria competitive Markov decision process

Курсова робота

за спеціальністю „Системний аналіз”

Керівник курсової роботи

к.н., доц. Чорней Р.К..

_____

(підпис)

“____” _____ 2021 р.

Виконала студентка Левченко І.С.

“____” _____ 2021 р.

Київ 2021

Content

Introduction

The course work is devoted to A multicriteria competitive Markov decision process; proposed software implementation of their solution.

The work consists of an introduction, the main part that consists of six sections, a conclusion, a list of used sources and an appendix.

Relevance. The modern-day world makes people face more and more complicated problems which require a solution and the price of mistake for them can be really high. Besides that, nowadays there is so much data that making a decision based on that intuitively and without analysis and math is not an option anymore. The multicriteria Markov decision process is much more similar to real-life than some other common games and decision models – choosing one of the available actions without knowing action chosen by the opponent as well as having vector reward rather than single reward are both much more common in a real application. However, solving such problems as they are is complicated. Therefore in this paper considered algorithm to transform them into linear programming problems, which have more well-known solution algorithms.

The object of the study is a multicriteria Markov decision process.

The subject of the study is an algorithm for solving the multicriteria competitive β-discounted Markov decision model.

Purpose to study multicriteria competitive Markov decision games and algorithm to solve them.

Theoretical research methods were used in the study; information from various scientific sources is analyzed, compared and summarized.

1. Markov decision process

Markov decision processes (MDPs) are discrete-time stochastic state-transition sequential models with outcome partially random partially under the control of the decision-maker. In MDP state changes randomly reacting to the actions of decision-maker (also known as an actor). Each state defines immediate reward obtained by actor and probabilities of other transitions from state to state [7]. A Markov decision process is a Markov chain with a choice that comes from actions, and motivation which resulted from rewards [8]. Therefore, MDP shares Markov property: The future is independent of the past given the present.

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1 \dots S_t]$$

Having the current state, previous states are irrelevant.

A Markov decision process problem consists of the state space, the set of actions, the cost or immediate reward function, and the set of transition probabilities. The objective is to minimize expected accumulated costs or maximize the expected accumulated rewards. A solution to MDP, also called Policy [3], is a correspondence of each state to an action to be taken in this state.

Markov decision processes have a finite number of states and actions that can be made by each actor in each state. Usually, it's assumed that perfect knowledge of what state the actor is in is available. [3] Perfect knowledge is unrequired in the case of partially observable Markov decision process problems.

In this paper, we generalize Markov decision processes to stochastic games and consider only the case with two decision-makers.

2. Multiple noncomparable criteria vector

In this paper, we consider games with multiple noncomparable criteria vector. Such games gain more interest nowadays as they better represent real-life situations [2][12]. Competitive games that can be modelled as a scalar zero-sum game can also be represented as a multicriteria zero-sum game that concurrently compares different scenarios. In such games not only a conflict of interest between

players appear but also for each individual. [9] Multiple noncomparable criteria vector would for example represent resources and/or gain and/or losses of the company on the competitive market: money, human resources, trust level among customers, market share etc. Therefore these resources cannot be compared and would be represented as a vector.

As we are working with vector standard solution approaches of scalar games cannot be used (optimization of one element of the vector could lead to a worse scenario for other elements). A number of new approaches were proposed, one of the popular ones including the concept of Pareto-optimality, to be precise of Pareto-optimal security strategy (POSS) [4]. Multi-objective optimization also known as Pareto optimization. As the solution of games considered in this paper, we consider a set of all POSS.

$x^* \in X$ is called Pareto optimal solution if and only if there are no other $x \in X$ such that $f_i(x) \leq f_i(x^*)$ for all $i = 1, \dots, k$, and $f_i(x) < f_i(x^*)$ for at least one i. Pareto optimal solution also called a non-inferior solution or nondominated. There is also exist the concept of Weak Pareto optimal solution. $x^* \in X$ is called Weak Pareto optimal solution if and only if there does not exist other $x \in X$ such that $f_i(x) < f_i(x^*)$ for at least one i. [10]

Pareto frontier, also known as Pareto front or Pareto set, is the set of allocations that are all Pareto efficient. Or as defined in [1] “The non-dominated set of the entire feasible search space S is the globally Pareto-optimal set”.

There are two common methods to calculate the Pareto front [6]:

1. presenting the multi-objective optimization problem as a series of single-objective optimization problems;
2. apply evolutionary methods which evolve solution from initial guess to the Pareto set.

Pareto-optimal security strategies (POSS) were introduced by Ghose and Prasad [5] as a solution concept in multicriteria zero-sum matrix games.

3. Basic notation

We observe the process in discrete time points. Process observed at a certain point of time called **state** and discrete moments of time **stages** [9]. S_t – is a random variable representing the state observed at the point of time t (t = 1, 2, 3, 4, …), that takes values from set $S = \{1, \dots, N\}$. Set S is finite, and s – represents the value from set S.

We refer to two decision-makers as to player 1 and player 2. For each player, there are a set of actions they can choose from at each state and a vector of rewards for each of the players that depend on the state and action are chosen by each player: player 1 choosing action $a^1 \in A^1(s)$ and player 2 choosing action $a^2 \in A^2(s)$ will result in player 1 receiving a reward vector $(r_1^1(s, a^1, a^2), \dots, r_k^1(s, a^1, a^2))$ and player 2 receiving reward vector $(r_1^2(s, a^1, a^2), \dots, r_k^2(s, a^1, a^2))$ [9]. Both players receive k different rewards (k – length of reward vector).

Stationary transition probabilities are the probability of the game moving to a certain state given the current state and action chosen. It is being assumed that it is a single controller case, meaning that stationary transition probabilities depend only on the action of one player [9].

$$P(s'|s, a^1) = P(S_{t+1} = s'|S_t = s, A_t^1 = a^1)$$

4. Strategy and rewards

Each player has a set of stationary strategies denoted by F_S for player 1 and G_S for player 2. Then we have $f = (f(1), \dots, f(N)) \in F_S$, where each $f(s)$ is a probability vector which dimension is equal to a number of actions that can be made by player 1 in state s ($|A^1(s)|$) and marked $m^1(s)$. Obviously, the sum of all elements of $f(s)$ equals 1.

$$f(s) := \left(f(s, 1), \dots, f\left(s, m^1(s)\right)\right)$$

$f(s, a^1)$ is a probability of player 1 choosing in state s action a^1 [9].

Similarly, we will have, $g = (g(1), \dots, g(N)) \in G_S$, $g(s) := \left(g(s,1), \dots, g\left(s, m^2(s)\right)\right)^T$ (T meaning transposed), with a sum of all elements of $g(s)$ equals 1.

$g(s, a^2)$ is a probability of player 2 choosing in state s action a^2.

A strategy called **pure** or **deterministic** if the probability of player choosing one particular action in this state equal to 1 ($f(s, a^1) \in \{0,1\}$ for all $a^1 \in A^1(s)$, $s \in S$ ($g(s, a^2) \in \{0,1\}$ for all $a^2 \in A^2(s)$, $s \in S$)). This means that deterministic (pure) control always selects one action (a_s^j, j=1,2) for state s. **A probability transition matrix** is a matrix of probabilities to move to a certain state given the current state and strategy. $P(s'|s, f) = \sum_{a^1=1}^{m^1(s)} f(s, a^1) p(s'|s, a^1) := f(s) P(s'|s)$ (sum of probabilities to choose action given state multiplied by probability to move to a certain state, given current state and action) [9].

Reward vector at time t player 1 is denoted as $(R_{1,t}^1, \dots, R_{k,t}^1)$ and for player 2 as $(R_{1,t}^2, \dots, R_{k,t}^2)$. We consider two denoting for immediate expected reward (IER):

- IER for the strategy pair $(f, g) \in F_S \times G_S$ at state s denoted as $r_1^1(s, f, g), \dots, r_k^1(s, f, g))$ for player 1 and $(r_1^2(s, f, g), \dots, r_k^2(s, f, g))$ for player 2.
- And we already considered IER vectors at state s with a^1 and a^2 action is chosen by players - $(r_1^1(s, a^1, a^2), \dots, r_k^1(s, a^1, a^2))$ and $(r_1^2(s, a^1, a^2), \dots, r_k^2(s, a^1, a^2))$. [9]

As the games we consider zero-sum games, it means that what one player gains another player loses and therefore we denotes $r_1(s, a^1, a^2) := r_1^1(s, a^1, a^2) = -r_1^2(s, a^1, a^2)$.

Considering expected rewards in the state s for one of the strategies we get

$$r_l(s, f, a^2) = \sum_{a^1=1}^{m^1(s)} f(s, a^1) r_l(s, a^1, a^2) := f(s) R_l(s, a^2), \quad (1)$$

and

$$r_l(s,a^1,g)=\sum_{a^2=1}^{m^2(s)} r_l(s,a^1,a^2)g(s,a^2) \coloneqq R_l(s,a^1)g(s). \qquad (2)$$

Therefore expected reward in state s for a pair of strategies (f,g) would be

$$r_l(s,f,g)=\sum_{a^1=1}^{m^1(s)}\sum_{a^2=1}^{m^2(s)} f(s,a^1)r_l(s,a^1,a^2)g(s,a^2) \coloneqq f(s)R_l(s)g(s)$$

Vector of l-th reward for all states is $r_l(f,g)=\big(r_l(1,f,g),\dots,r_l(N,f,g)\big)^T$.

5. Multicriteria β-discounted Markov decision model

As rewards are received at different point of times we need to account for discounting of values of such rewards. The model of such game described in this chapter was presented in 'A multicriteria competitive Markov decision process' article [9].

Let the sequence of random reward vectors for the period from stage till next stage $[t,t+1)$ be denoted as $\{R_t\}_{t=0}^{\infty}=\{(R_{1,t},\dots,R_{k,t})\}_{t=0}^{\infty}$. The probability of distribution of R_t for each t is specified when state and strategies are known.

We can define expected reward in criterion l, at stage t, state s, applying strategies f,g.

$$E_{sfg}\big(R_{l,t}\big)=\sum_{s'=1}^{N} p_t(s'|s,f)r_l(s',f,g) \coloneqq P_t(s,f)r_l(f,g)$$

$p_t(s'|s,f)r_l$ – is transition probability from state $\boldsymbol{s}$ into state $\boldsymbol{s'}$ on step $\boldsymbol{t}$ in the Markov chain.

The **overall discounted reward value** for $l=1,\dots,k$, for a strategy pair $(f,g)\in F_S\times G_S$ with initial state s is

$$v_{\beta,l}(s,f,g) \coloneqq \sum_{t=0}^{\infty}\beta^t\, E_{sfg}\big(R_{l,t}\big), \quad \forall l=1,\dots,k, \forall s\in S, \beta\in[0,1)$$

which is the element of the following vector.

$$v_{\beta,l}(f,g) = \left(v_{\beta,l}(1,f,g), \dots, v_{\beta,l}(N,f,g)\right)^T$$

β – is called the **discount factor**.

Having defined discount rewards we can now redefine MDP. The **multicriteria competitive β-discounted Markov decision process** for initial state s and strategy pair $(f,g) \in F_S \times G_S$ is the model using vector $(v_{\beta,1}(f,g), \dots, v_{\beta,k}(f,g))$ as a criterion and denoted as Γ_β or $Game\ \Gamma_\beta$ [9].

Using notation just introduced the ideal solution would be strategy pair (f^*, g^*) such that $v_{\beta,l}(f,g^*) \leq v_{\beta,l}(f^*,g^*) \leq v_{\beta,l}(f^*,g)$ for any $l = 1,..k$ and any strategies f and g which are not equal f^* and g^* respectively. Such an ideal pair of course may not exists as $v_{\beta,l}$ is a vector.

To define another idea of the solution concept of **security levels** has to be introduced.

Every strategy $f \in F_S$ defines the **lower security level $\underline{v}_{\beta,l}(s,f)$** for all states as the payoff with respect to every criterion $v_{\beta,l}(s,f,g), l = 1, \dots k$, when player 1 bets to maximize criterion [9]. Security level $\underline{v}_{\beta,l}(s,f)$ represents the minimal reward of player 1 who choose strategy f.

$$\underline{v}_{\beta,l}(s,f) = \min_{g \in G_S} v_{\beta,l}(s,f,g)$$

Every strategy $g \in G_S$ defines the **upper-security level $\overline{v}_{\beta,l}(s,g)$** for all states as the payoff with respect to every criterion $v_{\beta,l}(s,f,g), l = 1, \dots k$, when player 2 bets to minimize criterion. Security level $\overline{v}_{\beta,l}(s,g)$ represents a maximum loss of player 2 who choose strategy g.

$$\overline{v}_{\beta,l}(s,g) = \max_{f \in F_S} v_{\beta,l}(s,f,g)$$

As the security level is now defined, we can use it to search for solutions: maximize minimum reward or minimize maximum loss.

$$\underline{v}_{\beta,l}(s,f^*) = \max_{f \in F_S} \underline{v}_{\beta,l}(s,f)$$

$$\overline{v}_{\beta,l}(s,g^*) = \min_{g \in G_S} \overline{v}_{\beta,l}(s,g)$$

Strategies f^* and g^* here are Pareto-optimal solutions as we optimize vector. Hence

$$\underline{v}_{\beta}(f) = \left(\underline{v}_{\beta,1}(f), \dots, \underline{v}_{\beta,k}(f)\right)$$

$$\overline{v}_{\beta}(g) = \left(\overline{v}_{\beta,1}(g), \dots, \overline{v}_{\beta,k}(g)\right),$$

where

$$\underline{v}_{\beta,l}(f) = \left(\underline{v}_{\beta,l}(1,f), \dots, \underline{v}_{\beta,l}(N,f)\right)^T, l = 1, \dots, k$$

$$\overline{v}_{\beta,l}(g) = \left(\overline{v}_{\beta,l}(1,g), \dots, \overline{v}_{\beta,l}(N,g)\right)^T, l = 1, \dots, k.$$

Note that if the ideal solution (f^*, g^*) to the game exist

$$v_{\beta}(s,f^*,g^*) = \underline{v}_{\beta}(s,f^*) = \overline{v}_{\beta}(s,g^*).$$

6. Solving a multicriteria Markov decision model

First, we will redefine POSS using notations defined above.

A strategy $f^* \in F_S$ is Pareto optimal security strategy for player 1 if there are no $f \in F_S$ so $\bar{v}_{\beta}(f^*) \geq \bar{v}_{\beta}(f)$, *[there is at least one f'] so* $\bar{v}_{\beta}(f^*) \neq \bar{v}_{\beta}(f')$. [9]

A strategy $g^* \in G_S$ is Pareto optimal security strategy for player 2 if there is no g $\in G_S$ so $\bar{v}_{\beta}(g^*) \geq \bar{v}_{\beta}(g)$, *[there is at least one g'] so* $\bar{v}_{\beta}(g^*) \neq \bar{v}_{\beta}(g')$.

From [11], if decision space is compact and criterion vectors are continuous POSS exist.

Theorem proven in [9] states that The Pareto-solution set of Problem presented bellow coincides with the Pareto-optimal security strategies set of $Game\ \Gamma_\beta$.

Problem*:

$$\min \sum_{s=1}^{N} v_1(s), \dots, \sum_{s=1}^{N} v_k(s)$$

$$v_l(s) \geq R_l(s, a^1) g(s) + \beta P(s, a^1) v_l$$

$$\sum_{a^2 \in A^2(s)} g(s, a^2) = 1$$

$$g(s, a^2) \geq 0$$

$$v_l = \left(v_l(1), \dots, v_l(N)\right)^T$$

$$\forall s \in S, \forall a^1 \in A^1(s), \forall a^2 \in A^2(s), l = 1, \dots, k$$

This theorem is very important for multicriteria Markov decision processes represented by $Game\ \Gamma_\beta$; it allows to obtain a solution set by solving a multicriteria linear programming problem, which is much more studied.

Method of solving such games and code developed for such purpose will be applied to the following example.

Let there be two states with rewards vectors of length 2 and each player can choose between two actions.

$$S = \{1,2\}$$

$$A^1(s) = A^2(s) = \{1,2\}, \qquad for\ s \in S$$

$$\beta = 0.7$$

There are the following rewards.

$$\left(R_1(1), R_2(1)\right) = \begin{pmatrix} (10,6) & (-6,4) \\ (-4,0) & (8,3) \end{pmatrix}$$

$$\left(R_1(2), R_2(2)\right) = \begin{pmatrix} (-2,0) & (5,3) \\ (4,2) & (-10,-10) \end{pmatrix}$$

Elements of the matrixes $(R_{a^2=1}(s), R_{a^2=2}(s))$ of the size 2×2 are reward vectors $(r_{l=1}(s,a^1,a^2), r_{l=1}(s,a^1,a^2))$ with different values of s, a^1, a^2 as elements. The matrixes above representing the following.

$$\left(R_1(1), R_2(1)\right) = \begin{pmatrix} (r_1(1,1,1), r_2(1,1,1)) & (r_1(1,1,2), r_2(1,1,2)) \\ (r_1(1,2,1), r_2(1,2,1)) & (r_1(1,2,2), r_2(1,2,2)) \end{pmatrix}$$

$$\left(R_1(2), R_2(2)\right) = \begin{pmatrix} (r_1(2,1,1), r_2(2,1,1)) & (r_1(2,1,2), r_2(2,1,2)) \\ (r_1(2,2,1), r_2(2,2,1)) & (r_1(2,2,2), r_2(2,2,2)) \end{pmatrix}$$

There are the following transition data.

$$P(1) = (p(s'|1,a^1)^{2,\ \ 2}_{a^1=1,s'=1} = \begin{pmatrix} 0.5 & 0.5 \\ 0.8 & 0.2 \end{pmatrix}$$

$$P(2) = (p(s'|2,a^1)^{2,\ \ 2}_{a^1=1,s'=1} = \begin{pmatrix} 0.3 & 0.7 \\ 0.9 & 0.1 \end{pmatrix}$$

P(1) – is probability in state $s = 1$, move to a state $s' = \{1,2\}$, if player 1 choose action $a^1 = \{1,2\}$. For example, 0.3 is a probability to move from state $s = 2$ to state $s' = 1$ if player 1 choose action $a^1 = 1$; and 0.7 is a probability to move from state $s = 2$ to state $s' = 2$ if player 1 choose the same action $a^1 = 1$.

These matrixes are the input for the program.

```
beta=0.7
R11R21=[[[10,6],[-6,4]],[[-4,0],[8,3]]]
R12R22=[[[-2,0],[5,3]],[[4,2],[-10,-10]]]
P1=[[0.5,0.5],[0.8,0.2]]
P2=[[0.3,0.7],[0.9,0.1]]
```

Next, reformulate the Problem* using the data above.

$$\min \sum_{s=1}^{2} v_1(s), \sum_{s=1}^{2} v_2(s)$$

$$v_l(s) \geq R_l(s,a^1)g(s) + \beta P(s,a^1)v_1 \quad (using\ formula(2)) =>$$

$$v_l(s) \geq \sum_{a^2=1}^{2} r_l(s,a^1,a^2)g(s,a^2) + \beta P(s,a^1)v_l \;\Rightarrow$$

$$v_l(s)_{a^1=1}^{2} \geq \sum_{a^2=1}^{2} r_l(s,a^1,a^2)g(s,a^2) + \beta \sum_{s'=1}^{2} p(s'|s,a^1)v_l$$

$$\sum_{a^2 \in A^2(s)} g(s,a^2) = 1$$

$$g(s,a^2) \geq 0$$

$$v_1 = \left(v_1(1), \dots, v_l(N)\right)^T$$

$$\forall s \in \{1,2\}, \forall a^1 \in \{1,2\}, \forall a^2 \in \{1,2\}, l = 1,2$$

After inserting all the data in this formulation we get the following

$$\min(v_1(1) + v_1(2), v_2(1) + v_2(2))$$

$$v_1(1) \geq 10g(1,1) - 6g(1,2) + 0.7(0.5v_1(1) + 0.5v_1(2))$$

$$v_1(1) \geq -4g(1,1) + 8g(1,2) + 0.7(0.8v_1(1) + 0.2v_1(2))$$

$$v_2(1) \geq 6g(1,1) + 4g(1,2) + 0.7(0.5v_2(1) + 0.5v_2(2))$$

$$v_2(1) \geq 0g(1,1) + 3g(1,2) + 0.7(0.8v_1(1) + 0.2v_1(2))$$

$$v_1(2) \geq -2g(2,1) + 5g(2,2) + 0.7(0.3v_1(1) + 0.7v_1(2))$$

$$v_1(2) \geq 4g(2,1) - 10g(2,2) + 0.7(0.9v_1(1) + 0.1v_1(2))$$

$$v_2(2) \geq 0g(2,1) + 3g(2,2) + 0.7(0.3v_1(1) + 0.7v_1(2))$$

$$v_2(2) \geq 2g(2,1) - 10g(2,2) + 0.7(0.9v_1(1) + 0.1v_1(2))$$

$$g(1,1) + g(1,2) = 1$$

$$g(2,1) + g(2,2) = 1$$

$$g(1,1), g(1,2), g(2,1), g(2,2) \geq 0$$

We can clearly see that this problem is now a multicriteria linear programming problem.

To deal with vector the scalarization technic is used for the program. This technic uses the approach of combining multiple objective functions into one scalar objective function.

$$\min \sum_{s=1}^{2} v_1(s), \sum_{s=1}^{2} v_2(s)$$

$$\min\ \alpha\left(\sum_{s=1}^{2} v_1(s)\right) + (1-\alpha)\left(\sum_{s=1}^{2} v_2(s)\right), \alpha \in [0,1]$$

$$\min\ \alpha * v_1(1) + \alpha * v_1(2) + (1-\alpha) * v_2(1) + (\alpha) * v_2(2)), \alpha \in [0,1]$$

Going through a range of possible values of α - [0,1] allows us to find the whole Pareto front.

```
alfa=0
while alfa<=1:
    objectiveFunction(matrix,(str(alfa)+','+str(alfa)+','+str(1-alfa)+','+str(1-alfa)+',0,0,0,0,0'))
    print(alfa)
    print(minimise(matrix))
    alfa=alfa+e
```

Now we have a standard linear programming problem that can be solved with any of the standard methods. The program uses the simplex method. The whole program is in Appendix.

The solution to the problem is the following Pareto-optimal solutions.

$v_1(1) = 6.86;\ v_1(2) = 7.11;\ v_2(1) = 12.31;\ v_2(2) = 8.58;\ g(1,1) = 0.5;\ g(1,2) = 0.5;\ g(2,1) = 0.4;\ g(2,2) = 0.6;$

$v_1(1) = 44.62;\ v_1(2) = 83.09;\ v_2(1) = 9.99;\ v_2(2) = 7.13;\ g(1,1) = 0;\ g(1,2) = 1;\ g(2,1) = 0.48;\ g(2,2) = 0.52;$

$v_1(1) = 10.46;\ v_1(2) = 8.32;\ v_2(1) = 11.76;\ v_2(2) = 8.24;\ g(1,1) = 0.38;\ g(1,2) = 0.62;\ g(2,1) = 0.42;\ g(2,2) = 0.58;$

Such an approach allows solving Multicriteria Markov decision models in a quite simple way; however, it still has its own difficulties. Even small problem considered above, with only two actions available for players and rewards vectors of length two results in a rather big linear programming problem with 8 variables and 12 constraints. Moreover, such a problem has to be solved more than once for different α values in an objective, which takes a lot of time.

Conclusion

Course work was devoted to the study of A multicriteria competitive Markov decision process.

In the first section of the work, a concept and definition of the Markov decision process were introduced.

The second section covered basic concepts associated with Multiple noncomparable criteria vector.

The third section of the work describes the basic notation used in the paper.

The fourth section covers Concepts of strategy and rewards.

The fifth section introduces a β-discounted model of the multicriteria MDM and covers the concept of security levels.

The sixth section describes the algorithm of solving the problem properly defined in the previous section with an example.

References

1. Akbari, M., et al. “Artificial Neural Network and Optimization.” *Advances in Friction-Stir Welding and Processing*, 2014, pp. 543–599.

2. Bergstresser, K., and P. L. Yu. “Domination Structures and Multicriteria Problems in n-Person Games.” Theory and Decision, vol. 8, no. 1, 1977, pp. 5–48.

3. Edelkamp, Stefan, and Stefan Schrödl. “Introduction.” *Heuristic Search*, 2012, pp. 3–46.

4. Fernandez, F. R., and J. Puerto. “Vector Linear Programming in Zero-Sum Multicriteria Matrix Games.” *Journal of Optimization Theory and Applications*, vol. 89, no. 1, 1996, pp. 115–127.

5. Ghose, D., and U. R. Prasad. “Solution Concepts in Two-Person Multicriteria Games.” *Journal of Optimization Theory and Applicatio*ns, vol. 63, no. 2, 1989, pp. 167–189.

6. Keßler, Tobias, et al. “Use of Predictor Corrector Methods for Multi-Objective Optimization of Dynamic Systems.” *Computer Aided Chemical Engineering*, Edited by Kravanja Zdravko and Miloš Bogataj, vol. 38, 2016, pp. 313–318.

7. Littman, M.L. “Markov Decision Processes.” *International Encyclopedia of the Social & Behavioral Sciences*, 2001, pp. 9240–9242.

8. Marinescu, Dan C. “Big Data, Data Streaming, and the Mobile Cloud.” *Cloud Computing*, 2nd ed., Elsevier, 2018, pp. 439–487.

9. Rodrı´guez-Chı´a, A. M., et al. “A Multicriteria Competitive Markov Decision Process.” *Mathematical Methods of Operations Research*, vol. 55, no. 3, 2002, pp. 359–369.

10. Sakawa, Masatoshi. “Fuzzy Multiobjective and Multilevel Optimization.” *Multiple Criteria Optimization: State of the Art Annotated Bibliographic Surveys*, edited by Matthias Ehrgott and Xavier Gandibleux, vol. 52, Springer US, 2003, pp. 171–226.

11. Sawaragi, Y., et al., editors. “Theory of Multiobjective Optimization.” *Mathematics in Science and Engineering*, vol. 176, 1985.

12. Szidarovszky, Ferenc, et al. *Techniques for Multiobjective Decision Making in Systems Management*. Elsevier, 1986.

Appendix. Code for solving multicriteria β-discounted Markov decision model

```
import numpy as np
S=[1,2]
A1=[1,2]
A2=[1,2]
beta=0.7
R11R21=[[[10,6],[-6,4]],[[-4,0],[8,3]]]
R12R22=[[[-2,0],[5,3]],[[4,2],[-10,-10]]]
P1=[[0.5,0.5],[0.8,0.2]]
P2=[[0.3,0.7],[0.9,0.1]]
def generateMatrix(variables,constraints):
    #generates empty matrix of correct size
    matrix = np.zeros((constraints+1, variables+constraints+2))
    return matrix

def generateVariables(matrix):
    nOfColumns = len(matrix[0,:])
    nOfRows = len(matrix[:,0])
    var = nOfColumns - nOfRows -1
    variables = []
    for i in range(var):
        variables.append('x'+str(i+1))
    return variables

def checkBottom(matrix):
    #check bottom row for negative values
    nOfRows = len(matrix[:,0])
    m = min(matrix[nOfRows-1,:-1])
    if m>=0:
        return False
    return True
```

```
def findPivotToRemove(matrix):
    #what pivot with negative in right column to remove
        total = []
        nOfColumnsM = len(matrix[0,:])
        mm = min(matrix[:-1,nOfColumnsM-1])
        rowOfmM = None
        if mm<=0:
            rowOfmM = np.where(matrix[:-1,nOfColumnsM-1] == mm)[0][0]
        rowWithNegative=rowOfmM
        row = matrix[rowWithNegative,:-1]
        minimal = min(row)
        columnOfMinimalinRow = np.where(row == minimal)[0][0]
        col = matrix[:-1,columnOfMinimalinRow]
        #  find smallest ratio in column which is positive
        for a, i in zip(col,matrix[:-1,-1]):
            tempp=i/a
            if a**2>0 and tempp>0:
                temp=i/a
                total.append(temp)
            else:
                total.append(0)
        maxValueToReturn = max(total)
        for t in total:
            if t > 0:
                if t < maxValueToReturn:
                    element = t

        index = total.index(maxValueToReturn)
        return [index,columnOfMinimalinRow]



def findPivot(matrix):
    if checkBottom(matrix):
        total = []
```

```
        nOfRowsM = len(matrix[:,0])
        mm = min(matrix[nOfRowsM-1,:-1])
        rowOfmM = None
        if mm<=0:
            rowOfmM = np.where(matrix[nOfRowsM-1,:-1] == mm)[0][0]

        n = rowOfmM
        for a,b in zip(matrix[:-1,n],matrix[:-1,-1]):
            tempp=b/a
            if a**2>0 and tempp>0:
                total.append(tempp)
            else:
                total.append(0)
        element = max(total)
        for t in total:
            if t > 0 and t < element:
                element = t

        index = total.index(element)
        return [index,n]


def pivot(row,col,matrix):
    #pivot matrix removing negatives from last row and column
    nOfRows = len(matrix[:,0])
    nOfColumns = len(matrix[0,:])
    temp = np.zeros((nOfRows,nOfColumns))
    roww = matrix[row,:]
    if matrix[row,col]**2>0:
        e = 1/matrix[row,col]
        r = roww*e
        for i in range(len(matrix[:,col])):
            a = matrix[i,:]
            b = matrix[i,col]
            if list(a) == list(roww):
```

```
            continue
        else:
            temp[i,:] = list(a-r*b)
    temp[row,:] = list(r)
    return temp
print('Pivot cannot be done')


def canConstrainBeAdded(matrix):
    nOfRows = len(matrix[:,0])
    #if at least 2 zero rows
    empty = []
    for i in range(nOfRows):
        summ = 0
        for k in matrix[i,:]:
            summ += k**2
        # append zero if all elements in a row - zero
        if summ == 0:
            empty.append(summ)
    if len(empty)>1:
        return True
    return False

def constrain(matrix,equation):
    if canConstrainBeAdded(matrix) == True:
        nOfColumns = len(matrix[0,:])
        nOfRows = len(matrix[:,0])
        count = 0
        while count < nOfRows:
            rowCheck = matrix[count,:]
            sumRow = 0
            for i in rowCheck:
                sumRow=sumRow+  float(i**2)
            if sumRow == 0:
                # first row with all zero found
```

```
                row = rowCheck
                break
            count=count+1
        equation = equation.split(',')
        if 'G' in equation:
            g = equation.index('G')
            del equation[g]
            equation = [float(i)*-1 for i in equation]
        if 'L' in equation:
            l = equation.index('L')
            del equation[l]
            equation = [float(i) for i in equation]
        i = 0
        while i<len(equation)-1:
            row[i] = equation[i]
            i +=1
        row[-1] = equation[-1]
        row[var+j] = 1
    print('Constraint cannot be added')


def canAddObjective(matrix):
    #check if it possible to add objective function
    nOfRows = len(matrix[:,0])
    empty = []
    for i in range(nOfRows):
        summ = 0
        for j in matrix[i,:]:
            summ += j**2
        if summ == 0:
            # if zero row
            empty.append(summ)
    if len(empty)==1:
        # one zero row
        return True
    return False
```

```
def objectiveFunction(matrix,equation):
    if canAddObjective(matrix)==True:
        equation = [float(i) for i in equation.split(',')]
        nOfRows = len(matrix[:,0])
        row = matrix[nOfRows-1,:]
        i = 0
        while i<len(equation)-1:
            row[i] = equation[i]*-1
            i +=1
        row[-2] = 1
        row[-1] = equation[-1]
    print('Finish adding constraints before adding the objective function')

def checkRight(matrix):
    #looks for negatives above last row in the right column
    minim = min(matrix[:-1,-1])
    if minim>= 0:
        return False
    return True

def minimise(matrix, output='summary'):
    matrix[-1,:-2] = [-1*i for i in matrix[-1,:-2]]
    matrix[-1,-1] = -1*matrix[-1,-1]
    while checkRight(matrix)==True:
        matrix = pivot(findPivotToRemove(matrix)[0],findPivotToRemove(matrix)[1],matrix)
    while checkBottom(matrix)==True:
        matrix = pivot(findPivot(matrix)[0],findPivot(matrix)[1],matrix)
    nOfColumns = len(matrix[0,:])
    nOfRows = len(matrix[:,0])
    var = nOfColumns - nOfRows -1
    i = 0
    values = {}
    for i in range(var):
```

```
            column = matrix[:,i]
            maxColumn = max(column)
            if float(sum(column)) == float(maxColumn):
                loc = np.where(column == maxColumn)[0][0]
                values[generateVariables(matrix)[i]] = matrix[loc,-1]
            else:
                values[generateVariables(matrix)[i]] = 0
        values['min'] = matrix[-1,-1]*-1
        for k,v in values.items():
            values[k] = round(v,6)
        if output == 'matrix':
            return matrix
        return values


    if __name__ == "__main__":
        matrix = generateMatrix(8,16)
        alfa=0
        constrain(matrix,(str((beta*(P1[0][0]))-
1)+','+str(beta*(P1[0][1]))+',0,0,'+str(R11R21[0][0][0])+','+str(R11R21[0][1][0])+',0,0,L,0'))
        constrain(matrix,(str((beta*(P1[1][0]))-
1)+','+str(beta*(P1[1][1]))+',0,0,'+str(R11R21[1][0][0])+','+str(R11R21[1][1][0])+',0,0,L,0'))
        constrain(matrix,('0,0,'+str((beta*(P1[0][0]))-
1)+','+str(beta*(P1[0][1]))+','+str(R11R21[0][0][1])+','+str(R11R21[0][1][1])+',0,0,L,0'))
        constrain(matrix,('0,0,'+str((beta*(P1[1][0]))-
1)+','+str(beta*(P1[1][1]))+','+str(R11R21[1][0][1])+','+str(R11R21[1][1][1])+',0,0,L,0'))


        constrain(matrix,(str((beta*(P2[0][0])))+','+str((beta*(P2[0][1]))-
1)+',0,0,0,0,'+str(R12R22[0][0][0])+','+str(R12R22[0][1][0])+',L,0'))
        constrain(matrix,(str((beta*(P2[1][0])))+','+str((beta*(P2[1][1]))-
1)+',0,0,0,0,'+str(R12R22[1][0][0])+','+str(R12R22[1][1][0])+',L,0'))
        constrain(matrix,('0,0,'+str((beta*(P2[0][0])))+','+str((beta*(P2[0][1]))-
1)+',0,0,'+str(R12R22[0][0][1])+','+str(R12R22[0][1][1])+',L,0'))
        constrain(matrix,('0,0,'+str((beta*(P2[1][0])))+','+str((beta*(P2[1][1]))-
1)+',0,0,'+str(R12R22[1][0][1])+','+str(R12R22[1][1][1])+',L,0'))


        constrain(matrix,('0,0,0,0,'+str(1)+',0,0,0,G,0'))
        constrain(matrix,('0,0,0,0,0,'+str(1)+',0,0,G,0'))
        constrain(matrix,('0,0,0,0,0,0,'+str(1)+',0,G,0'))
```

```
constrain(matrix,('0,0,0,0,0,0,0,'+str(1)+',G,0'))
constrain(matrix,('0,0,0,0,'+str(1)+','+str(1)+',0,0,G,1'))
constrain(matrix,('0,0,0,0,'+str(1)+','+str(1)+',0,0,L,1'))
constrain(matrix,('0,0,0,0,0,0,'+str(1)+','+str(1)+',G,1'))
constrain(matrix,('0,0,0,0,0,0,'+str(1)+','+str(1)+',L,1'))

e=0.1
alfa=0
while alfa<=1:
    objectiveFunction(matrix,(str(alfa)+','+str(alfa)+','+str(1-alfa)+','+str(1-alfa)+',0,0,0,0,0'))
    print(alfa)
    print(minimise(matrix))
    alfa=alfa+e
```