Desenho de Mecanismo

Ian Teixeira Barreiro

Fevereiro 2022

1 Introdução

A área de desenho de mecanismo pode ser vista como o problema inverso daquele enfrentado em outras áreas da teoria dos jogos. Isso por que, enquanto em jogos comuns a preocupação é com os resultados globais que serão atingidos quando um grupo de agentes age de maneira racional e auto-interessada em um jogo, a área de desenho de mecanismo parte de um objetivo global que se quer atingir e analise as regras que o jogo deve ter para que as ações dos agentes participantes do jogo levem a alcançar o objetivo global pretendido. Tipicamente, problemas de desenho de mecanismo envolvem situações em que há informação incompleta sobre o jogo, notadamente, os agentes podem ter incentivos para não revelar aos demais suas características. Nesse cenário, é do interesse do planejador criar mecanismos que levem os agentes racionais a informarem suas características de modo honesto. Dado que no cenário típico estamos em um jogo de informação incompleta, podemos considerar os problemas de desenho de mecanismo como um caso especial de jogos bayesianos.

No presente texto faremos um resumo do tópico de desenho de mecanismo. Usaremos como referências principais os textos "Game Theory: An Introduction"de Steven Tadelis e "Game Theory and Mechanism Design"de Y. Narahari. O texto de estruturará da seguinte maneira i) primeiro faremos uma descrição do problema de desenho de mecanismo; ii) depois descreveremos os problemas de desenho de mecanismo como problemas de encontrar regras para um jogo que levem para toda estratégia dos agentes ao objetivo pretendido; iii) então descreveremos o desenho de mecansimo como um problema de revelação de preferências; iv) depois descreveremos um exemplo de problema de desenho de mecanismo v) por fim apresentaremos uma implementação computacional que solucione o problema.

2 Desenho de Mecanismo: Estrutura Básica

A estrutura de um problema de desenho de mecanismo se compõe basicamente de um conjunto de N jogadores e X alternativas disponíveis para a escolha dos jogadores. Essas alternativas podem ser diversas coisas, no sentido que elas podem ser tanto um serviço que será prestado para um ou todos os jogadores,

ou um bem que está sendo leiloado aos jogadores ou uma decisão de investimento público (como investimento em educação ou saneamento) que impactará os jogadores. O conjunto de alternativas é considerado um conjunto de alternativas públicas pois a escolha por determinado $x \in X$ afeta todos os jogadores participando de um jogo (se decide-se não investir em saneamento, por exemplo, isso afeta todos os jogadores). Cada jogador pode ter um tipo diferente, $\theta_i \in \Theta_i$ em que Θ_i é o conjunto de todos os tipos possíveis para aquele jogador, e o tipo do jogador determina quais são suas preferências sobre os resultados do jogo. O tipo de cada jogador é uma informação privada para aquele jogador, que pode ou não ser conhecida pelos demais jogadores ou pelo planejador. O conjunto $\Theta = \Theta_1 \times \Theta_2 \times ... \times \Theta_n$ expressa todos os conjuntos de tipos possíveis que os jogadores podem assumir e é chamado de espaço de estados. Cada elemento desse conjunto é um estado de mundo possível. O estado de mundo que efetivamente se concretiza ocorre de acordo com uma função a priori $\phi(.)$ que é de conhecimento de todos os jogadores, e que é atualizada por informações particulares de cada agente. Com relação aos payoffs consideraremos que além da utilidade de cada agente por cada alternativa, os payoffs também são compostos por valores monetários auferidos de acordo com o resultado do jogo de modo que os payoffs totais podem ser expressos pela seguinte função quase-linear: $v(x, m_i, \theta_i) = u(x, \theta_i) + m_i$, em que m_i é a recompensa monetária.

Vamos agora olhar o problema da perspectiva do designer de mecanismo ou autoridade central. No caso dele, seu objetivo é alcançar algum resultado baseado nos tipos dos agentes. Assume-se que o planejador não tenha recursos para pagar as recompensas dos agentes, de modo que alguns agentes deverão pagar para os outros. Além disso, o planejador pode apropriar para si uma parcela das recompensas monetárias, de modo que os resultados do jogo podem ser expressos da maneira que segue.

$$Y = (x, m_1, ..., m_n) : x \in X, m_i \in \mathbb{R} \forall i \in N, \sum_{i=1}^{n} m_i \leq 0$$

O objetivo do planejador é dada por uma regra de escolha, que é uma função que para cada estado de mundo realiza uma escolha de resultado entre as alternativas e determina a recompensa monetária para cada agente.

$$f(\theta) = (x(\theta), m_1(\theta), m_2(\theta), ..., m_n(\theta))$$

Ocorre que a autoridade não sabe quais são os tipos dos jogadores e por isso não consegue por em prática sua função de escolha social. Há dois caminhos de solução possíveis. Primeiro, ele pode encontrar regras para um jogo que quando implementadas levem os agentes a revelarem suas preferências indiretamente. Segundo, ele poderia simplesmente perguntar aos agentes seus tipos, e implementar funções sociais que tornem mais atrativo aos agentes revelarem seus tipos verdadeiros.

3 Mecanismos Indiretos

A primeira solução descrita anteriormente para o problema do designer de mecanismo dá origem aos chamados mecanismos indiretos. Neles cada agente possui um conjunto de ações disponíveis A_i e uma função $g : A \to Y$ que mapeia cada conjunto de ações de cada agente a um resultado. Cada agente possui também uma função de estratégias $s_i : \Theta_i \to A_i$. Os equilíbrios de Nash nesse cenário se dão por conjuntos de funções de estratégias, tais que para cada agente o payoff esperado de tomar essa estratégia, dadas as estratégias que ele espera que os outros agentes tomem, é superior ao payoff de qualquer outra ação no mesmo cenário. Um equilíbrio de Nash implementa o mecanismo quando $g(s_1(\theta_i), s_2(\theta_2), ..., s_n(\theta_n)) = f(\theta), \forall\theta \in \Theta$.

4 Mecanismos Diretos: O Princípio da Revelação

A ideia dos mecanismos diretos é criar mecanismos em que as ações de cada um dos jogadores sejam revelar de maneira verdadeira qual o seu tipo. Esse tipo de mecanismo é implementável se a expectativa de ganho de cada jogador ao anunciar seu tipo verdadeiro, dada a forma que o jogador acredita que os demais jogadores irão agir, é maior do que anunciar qualquer outro tipo que não o seu verdadeiro. A autoridade central deverá, portanto, criar um mecanismo em que hajam incentivos para todos os jogadores revelarem seus tipos verdadeiros.

5 Exemplo de Problema

O problema básico que será analisado inicialmente é um em que um comprador deseja comprar um determinado produto e existem 2 vendedores possíveis para esse produto. O vendedor 1 possui ao seu dispor a tecnologia a_1 e o vendedor 2 possui ao seu dispor a tecnologia a_2, que é mais avançada que a do vendedor 1, e a tecnologia b_2, que é menos avançada que a do vendedor 1. Desse modo, temos que $\Theta = (a_1, a_2), (a_1, b_2)$ é o conjunto domínio da função de escolha social. O conjunto de alternativas é $X = x, y, z$, em que x é o cenário em que 100% dos bens são comprados do vendedor 1, y é o cenário em que 50% dos bens são comprados de cada vendedor e z é o cenário em que 100% da compra é feita com o vendedor 2. Para completar a descrição do problema temos que a função utilidade do vendedor 1 é $u_1(x, a_1) = 100, u_1(y, a_1) = 50, u_1(z, a_1) = 0$ e do vendedor 2 é $u_2(x, a_2) = 0, u_2(y, a_2) = 50, u_2(z, a_2) = 100, u_2(x, b_2) = 0, u_2(y, b_2) = 50, u_2(z, b_2) = 25$.

6 Implementação Computacional

```
1 def dominio(lista_thetas):
2
3     """
```

```
    Funcao cria o dominio de f
    """

    # Aplica o produto cartesiano entre os conjuntos
    # dos tipos de cada agente
    prod_cart = list(product(*lista_thetas))

    # Transforma a lista de tuplas em uma matriz
    dom = [[j for j in prod_cart[i]] for i in range(len(prod_cart))
    ]

    return np.array(dom)

def social_choice_functions(dominio, contradominio):

    """
    Cria todas as possibilidades de funcoes de escolha social
    """

    # Para cada elemento no dominio cria uma copia
    # da lista de escolhas publicas disponiveis
    aux = [contradominio for i in range(len(dominio))]

    # Faz o produto cartesiano entre cada uma das copias
    # da lista de escolhas publicas
    prod_cart = list(product(*aux))

    # Transforma a lista de tuplas em uma matriz
    scf = [[j for j in prod_cart[i]] for i in range(len(prod_cart))
    ]

    return np.array(scf)

def f_u2(choice, theta):

    """
    Funcao utilidade da pessoa 2
    """

    # Caso a tecnologia seja b2 e a escolha seja z
    if theta == 2 and choice == 3:

        return 25

    # Caso contrario
    else:

        if choice == 1:

            return 0

        elif choice == 2:

            return 50

        else:

```

```
            return 100

def utilidades(scf, f_util, thetas):

    """
    Recebe todas as funcoes de escolha social, a funcao utilidade
    de um agente e o conjunto thetas de tipos daquele agente.
    Retorna
    um tensor com a utilidade do agente para cada tipo e cada
    funcao
    """

    # Cria lista de utilidades - ha len(scf) listas, representando
    # cada funcao de escolha social possivel. Dentro de cada uma
    # dessas listas ha len(thetas) representando cada tipo de
    agente
    u = [[[] for j in range(len(thetas))] for i in range(len(scf))]

    # Para cada funcao de escolha social f
    for f in range(len(scf)):

        # Para cada tipo t do agente
        for t in range(len(thetas)):

            # Para cada escolha social feita, apensa a
            # utilidade do agente caso ele for do tipo t
            for c in scf[f]:

                u[f][t].append(f_util(c, thetas[t]))

    return u


def implementavel_facil(scf, thetas, utilidades):

    """
    Encontra as funcoes de escolha social implementaveis
    """

    # Cria a mascara preenchida com True
    implementavel = [True for i in range(len(scf))]

    # Para cada funcao de escolha social
    for f in range(len(utilidades)):

        # Para cada tipo do agente 2
        for t in range(len(thetas)):

            # Se as utilidades de cada tipo forem iguais
            # passa para a proxima iteracao - sera uma
            # implementacao valida para aquele tipo
            if utilidades[f][t][0] == utilidades[f][t][1]:

                break

            # Caso o maior payoff seja mentir, ou seja
            # seja dizer que voce eh de outro tipo que
```

```
113             # nao o seu verdadeiro, a funcao nao eh
114             # implementavel
115             elif np.argmax(utilidades[f][t]) != t:
116 
117                 implementavel[f] = False
118 
119     # Aplica a mascara no conjunto de todas as
120     # funcoes de escolha social
121     impl = scf[implementavel]
122 
123     return impl
```