Mechanism_Design

January 16, 2026

Author: Vaibhav Thakur

1 Mechanism Design Search Program

Goal : Find a Social Choice Function (SCF) that is Monotone but NOT Strategyproof.
Setup : 3 Alternatives, 2 Agents.

[ ]:
```
# We have three alternatives and two players

# Alternatives
alts = ['a','b','c'] # Possible outcomes

# All possible strict preference orderings are
preferences = [
    ['a','b','c'],
    ['a','c','b'],
    ['b','a','c'],
    ['b','c','a'],
    ['c','a','b'],
    ['c','b','a']
]
```

Functions

[ ]:
```
# B(a,Pi) : Below set for alternative a in Pi
def B(a,Pi):
    index = Pi.index(a)
    return Pi[index+1:]

# prefers(Pi,a,b) : True if a is ranked above b in profile Pi, False otherwise
def prefers(Pi, a, b):
    return Pi.index(a) < Pi.index(b)

# Set of all possible preferences,i.e., power set of preferences[]
def pref_possibilities():
    psbl_prefs = []
    for x in preferences:
        psbl_prefs = psbl_prefs + [[x]] + [s + [x] for s in psbl_prefs]
```

```
    return psbl_prefs
```

def pref_possibilities() is generating power set without the null set, because for an agent, allowed preferences can be any subset of the preferences as we are considering the case of both restricted and unrestricted domain.

Working : When we process elements one by one, every new element has two choices for every existing subset: either not included, or included So if we already have some subsets, adding a new element x means: 1. Keep all existing subsets as they are 2. create new subsets by adding x to each existing subset.

Now will generate all possible domains for the social choice function.

[ ]:
```
psbl_prefs = pref_possibilities()
domains = [(x,y) for x in psbl_prefs for y in psbl_prefs]
```

Monotonicity check

[ ]:
```
def is_monotone(f, profiles):
    for P in profiles:
        key_P = (tuple(P[0]), tuple(P[1]))
        a = f[key_P]
        for P_prime in profiles:
            key_P_prime = (tuple(P_prime[0]), tuple(P_prime[1]))
            if all(set(B(a, P[i])).issubset(set(B(a, P_prime[i]))) for i in␣
 ↪[0,1]):
                if f[key_P_prime] != a:
                    return False
    return True
```

Strategyproofness(SP) : A function is strategyproof if no agent can misreport their preference to get an outcome they strictly prefer.

Strategyproofness check : Checks if a social choice function(scf) is SP for a given list of preference profiles.

f : function taking a profile (P1,P2) and returning an outcome in {a,b,c} (we will use Python dictionary to define our function)

Returns : True if SP, False otherwise

[ ]:
```
def is_sp(f, profiles):

    # Loop over each agent: 0 = agent 1, 1 = agent 2
    for i in [0, 1]:

        # Go through every possible profile in the domain
        for P in profiles:
            key_P = (tuple(P[0]), tuple(P[1]))
```

```
            # The current (true) outcome when both report honestly
            true_outcome = f[key_P]

            # The current agent's true preference ordering
            true_pref = P[i]

            # Try every possible "lie" the agent could make
            for P_i_misreport in preferences:  # preferences = all 6 strict␣
↪orders
                # Construct a new profile where only agent i lies

                # Copy the original profile in P_misreport
                P_misreport = [P[0][:], P[1][:]]
                # Replace the preference of i by his misreported preference.
                P_misreport[i] = P_i_misreport[:]

                key_misreport = (tuple(P_misreport[0]), tuple(P_misreport[1]))

                # Check if misreported profile belongs to domain
                if key_misreport not in f:
                    continue
                new_outcome = f[key_misreport]
                if prefers(true_pref, new_outcome, true_outcome):
                    # Agent i benefits from misreporting -> not strategyproof
                    return False
    return True
```

Enumerate all possible social choice functions: Enumerates every possible deterministic scf for the given domain (list of profiles). Each scf is a rule f(P1,P2) -> outcome {a,b,c}.

We generate all possible ways to assign outcomes to all profiles.

Parameters: profiles : list of [P1, P2] All possible preference pairs in the domain. Example: [[[‘a’,‘b’,‘c’], [‘b’,‘a’,‘c’]], [[‘b’,‘a’,‘c’], [‘a’,‘b’,‘c’]], …]

Returns: functions (using Python dictionaries)

[ ]:
```
def enumerate_functions(profiles):
    alts = ['a', 'b', 'c']
    all_functions = []

    n = len(profiles)
    total = 3**n

    for j in range(total):
        assignment = []
        temp = j
        for k in range(n):
            choice_index = temp%3
```

```
            assignment.append(alts[choice_index])
            temp = temp//3

        mapping = { (tuple(profiles[i][0]), tuple(profiles[i][1])):␣
↪assignment[i] for i in range(n) }
        all_functions.append(mapping)

    # Outer loop accounts for all possible functions.
    # Inner loop gives mapping to each profile under the corresponding function␣
↪in the outer loop.
    return all_functions
```

Main Search

[ ]:
```
found = False
counter = 0

for (D1, D2) in domains:
    profiles = [(p1, p2) for p1 in D1 for p2 in D2]

    functions = enumerate_functions(profiles)
    print(f" Checking domain with {len(D1)}* {len(D2)} preferences␣
↪({len(profiles)})...")

    for f in functions:
        if is_monotone(f, profiles) and not is_sp(f,profiles):
            print('\n FOUND MONOTONE BUT NOT STRATEGYPROOF FUNCTION')
            print('Domain D1: ', D1)
            print('Domain D2: ', D2)
            print("Profiles:", len(profiles))
            print("Outcome table:")
            for P in profiles:
                key_for_P = (tuple(P[0]),tuple(P[1]))
                print(f"{P} → {f[key_for_P]}")
            found = True
            break
    if found:
        break

if not found:
    print('\n No monotone-but-not-strategyproof SCF found')
```