Representación Computacional y Búsqueda de Equilibrios de Nash en Juegos Estáticos de 2 Jugadores

En este documento abordamos cómo podemos **modelar** un juego estático de dos jugadores (información perfecta, acciones simultáneas, no cooperativo) de forma computacional, así como distintos **algoritmos** para encontrar **equilibrios de Nash en estrategias puras**. Incluiremos **pseudocódigo** y un breve **análisis de complejidad**, además de comentar la escalabilidad cuando el número de jugadores o estrategias crece.

---

1. Representación Computacional

1.1. Estructura de Datos

Para un **juego estático de 2 jugadores** con un conjunto finito de estrategias:

- **Jugador A**:
  - Estrategias: $S_A = {s_{A1}, s_{A2}, \dots, s_{A m}}$
- **Jugador B**:
  - Estrategias: $S_B = {s_{B1}, s_{B2}, \dots, s_{B n}}$

Las **utilidades** pueden representarse como dos matrices (o arreglos 2D) si cada jugador tiene m y n estrategias respectivamente:

1. $\text{PayoffA}[i][j]$ = Utilidad de A cuando A juega la estrategia i y B juega la estrategia j
2. $\text{PayoffB}[i][j]$ = Utilidad de B bajo la misma combinación de estrategias

Así, si A tiene m estrategias y B tiene n estrategias, tendremos dos matrices de tamaño $m \times n$.

PROF

Ejemplo en pseudocódigo:

```
// Supongamos que m,n son las cantidades de estrategias
// Creamos dos matrices de payoff:
PayoffA = new Matrix(m, n)
PayoffB = new Matrix(m, n)

// Rellenar con datos
for i in [0..m-1]:
    for j in [0..n-1]:
        PayoffA[i][j] = valorDouble  // asignado desde la definición del juego
        PayoffB[i][j] = valorDouble
```

- Esta representación es muy **natural** para juegos 2x2, 3x3 o en general (m x n).
- Alternativamente, se puede usar una sola estructura de datos (por ejemplo, una matriz de tuplas), pero normalmente se desdobla en dos, por legibilidad y conveniencia.

---

2. Algoritmos para Encontrar Equilibrios de Nash en Estrategias Puras

2.1. Definición Rápida de Equilibrio de Nash (en puras)

Un perfil de estrategias $(i^, *j^*)$ (donde i^* es una estrategia de A y j^* una estrategia de B) es **Equilibrio de Nash** si cumple:

- Dada la estrategia j^* de B, i^* es la **mejor respuesta** de A.
  $PayoffA[i^*][j^*] \geq PayoffA[i][j^*] \quad \text{para todo } i \in S_A$
- Dada la estrategia i^* de A, j^* es la **mejor respuesta** de B.
  $PayoffB[i^*][j^*] \geq PayoffB[i^*][j] \quad \text{para todo } j \in S_B$

2.2. Búsqueda Exhaustiva (Brute Force)

Idea: Verificar cada perfil (i, j) de estrategias y comprobar si cumple las condiciones de "mejor respuesta" para cada jugador.

1. Para cada estrategia i de A y cada estrategia j de B:
   - Revisar si, con j fijo, no hay otra estrategia i' que otorgue a A una utilidad mayor que $PayoffA[i][j]$.
   - Revisar si, con i fijo, no hay otra estrategia j' que otorgue a B una utilidad mayor que $PayoffB[i][j]$.
2. Si no existe tal i' ni j' que mejoren el pago, entonces (i, j) es un equilibrio de Nash (en puras).

Pseudocódigo:


```
function findPureStrategyNashEquilibria(PayoffA, PayoffB, m, n):
    equilibria = []

    for i in [0..m-1]:
        for j in [0..n-1]:

            // 1. Checar si A puede mejorar
            canAImprove = false
            for iPrime in [0..m-1]:
                if PayoffA[iPrime][j] > PayoffA[i][j]:
                    canAImprove = true
                    break

            // 2. Checar si B puede mejorar
            canBImprove = false
```

```
            for jPrime in [0..n-1]:
                if PayoffB[i][jPrime] > PayoffB[i][j]:
                    canBImprove = true
                    break

            if (not canAImprove) and (not canBImprove):
                equilibria.push( (i, j) )

    return equilibria
```

Complejidad

- Se itera sobre $m \times n$ perfiles.
- Para cada uno, se hacen dos bucles de comparación (uno de tamaño m y otro de tamaño n).
- **Complejidad total**: $O(m \times n \times (m + n))$ = $O(m^2 \times n + m \times n^2)$.
  - Para juegos pequeños (2x2, 3x3, etc.) esto es muy manejable.
  - Si m y n crecen mucho, el costo crece de forma **cúbica** en el peor caso.

2.3. Búsqueda Basada en "Mejores Respuestas"

Otra forma de pensar en un **método más directo** (aunque en esencia similar) es:

1. Para cada **columna** j, determina las estrategias de A que son **mejor respuesta** (todas las i que maximizan $PayoffA[i][j]$).
2. Para cada **fila** i, determina las estrategias de B que son **mejor respuesta** (todas las j que maximizan $PayoffB[i][j]$).
3. Un **equilibrio** ocurre cuando una pareja (i, j) está en la lista de mejores respuestas mutuas.

Pseudocódigo (en esquema simplificado):


```
// 1. Mejores respuestas de A para cada j
bestResponsesA = array of lists, size n  // para cada j, guardamos lista
de i's
for j in [0..n-1]:
    // Encontrar max payoff para A en esta columna
    maxForA = -∞
    for i in [0..m-1]:
        if PayoffA[i][j] > maxForA:
            maxForA = PayoffA[i][j]
    // Colectar todos los i que alcancen este valor
    for i in [0..m-1]:
        if PayoffA[i][j] == maxForA:
            bestResponsesA[j].add(i)

// 2. Mejores respuestas de B para cada i
bestResponsesB = array of lists, size m
for i in [0..m-1]:
    // Encontrar max payoff para B en esta fila
```

```
    maxForB = -∞
    for j in [0..n-1]:
        if PayoffB[i][j] > maxForB:
            maxForB = PayoffB[i][j]
    // Colectar todos los j que alcancen este valor
    for j in [0..n-1]:
        if PayoffB[i][j] == maxForB:
            bestResponsesB[i].add(j)

// 3. Intersecciones (i,j) que sean mejores respuestas mutuas
equilibria = []
for j in [0..n-1]:
    for i in bestResponsesA[j]:
        if j in bestResponsesB[i]:
            equilibria.push( (i, j) )
return equilibria
```

- **Complejidad**:
  - El paso 1 y el paso 2 cada uno toman $O(m \times n)$ (buscar máximo por fila/columna).
  - Luego, construir la lista final y verificar intersecciones también es $O(m \times n)$ en el peor caso.
  - En total, este método es $O(m \times n)$, **más eficiente** que el brute force comparando todas las desviaciones.

En la práctica, este método se prefiere cuando **solo** se buscan **equilibrios puros** y los tamaños de matrices son moderados.

---

3. Observaciones sobre la Escalabilidad

- El método de la **matriz de pagos** crece de **forma exponencial** si pensamos en **más** de 2 jugadores o si cada jugador tiene **muchas estrategias**.
  - Para **2 jugadores** con m y n estrategias, la representación es del orden $O(m \times n)$.
  - Para **N jugadores**, cada uno con k estrategias, la matriz en "forma normal" sería de tamaño k^N (pues hay que registrar el pago para cada combinación de k estrategias en N jugadores).
- **Complejidad computacional**: Encontrar un Equilibrio de Nash en un juego general (con más jugadores y muchas estrategias) puede ser **muy costoso**.
  - Para 2 jugadores y pocos m, n, la búsqueda es manejable con métodos de enumeración.
  - Para juegos grandes (o más jugadores), se utilizan algoritmos más avanzados (por ejemplo, **algoritmos de reducción** tipo Lemke-Howson para juegos bimatriz o métodos de programación lineal en el caso de equilibria mixtos).

---

4. Reflexión: Cuando Aumentan Jugadores o Estrategias

- **Número de jugadores (N)**:

  - Con $N > 2$, necesitamos un vector de **N utilidades** para cada combinación de estrategias $(s_1 \times s_2 \times \dots \times s_N)$. El espacio de combinaciones crece exponencialmente.
  - En la práctica, se busca **representaciones más compactas** (ej., juegos polimátricos, grafos de interacción) o se aplican técnicas de **algoritmos aproximados** o **heurísticas**.
- **Número de estrategias (k)**:
  - Aunque sigamos con 2 jugadores, cada uno con k estrategias, la matriz pasa a tener $k \times k$ celdas. Todavía es $O(k^2)$ en almacenamiento y el método de búsqueda de equilibrios puros sigue en $O(k^2)$ con la mejora de "mejores respuestas".
  - Si k es muy grande, se vuelve impráctico enumerarlas todas; a veces se usan métodos de **reducción de estrategias dominadas** o se focaliza en equilibria mixtos con metodologías específicas (ej.: **algoritmo de simplicial subdivision**, Lemke-Howson, etc.).

---

5. Conclusiones Principales

1. **Modelado Computacional**

   - Para dos jugadores, es straightforward usar dos matrices de tamaño $m \times n$.
   - Rellenar dichas matrices con las utilidades requiere conocer la definición del juego.

2. **Algoritmos para Equilibrio de Nash en Puras**

   - **Brute Force**: Comparar cada estrategia con sus posibles desvíos, complejidad $O(m^2 n + m n^2)$.
   - **Búsqueda de Mejores Respuestas**: Identificar mejores respuestas para cada fila/columna y buscar intersecciones, complejidad $O(mn)$.

3. **Escalabilidad**

   - Con **2 jugadores** y estrategias razonables, es **viable**.
   - El **aumento** en número de jugadores o estrategias hace que la representación y el cálculo de equilibria (incluso solo puros) se torne **exponencial** o muy costosa.

Por ello, en la práctica computacional de la teoría de juegos, uno se basa en:

- **Reducciones de complejidad** (buscando subfamilias de estrategias, eliminando dominadas).
- **Algoritmos especializados** (Lemke-Howson, etc. para equilibria mixtos).
- **Heurísticas** o métodos de **approx** en casos muy grandes.

Referencias Breves

- **Lemke-Howson** (1964): algoritmo clásico para juegos bimatriz (2 jugadores) que busca **equilibrio en estrategias mixtas**.
- **Programación Lineal** (Ej. Teorema Minimax) para juegos de suma cero.
- **Enumeración** pura de celdas, útil en entornos educativos o con matrices pequeñas.
