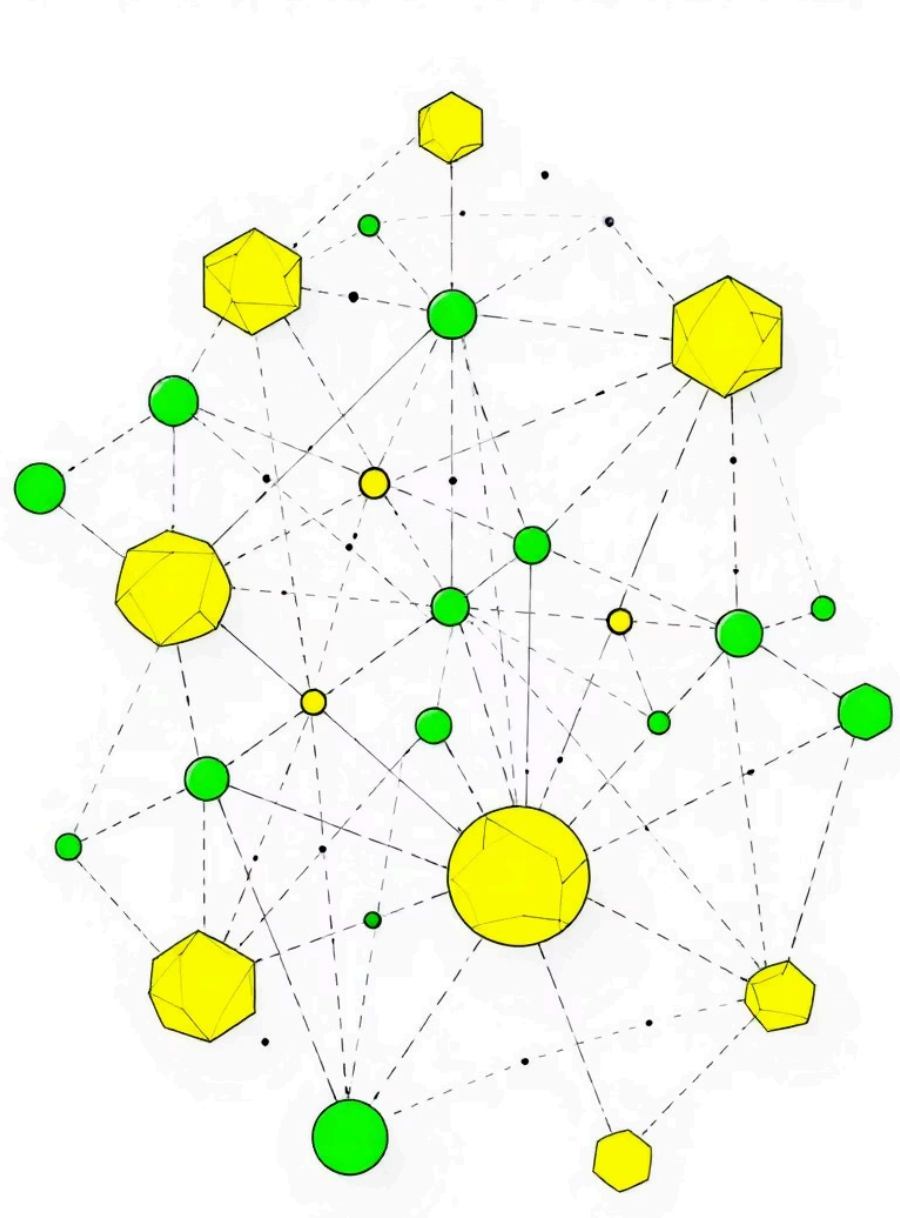
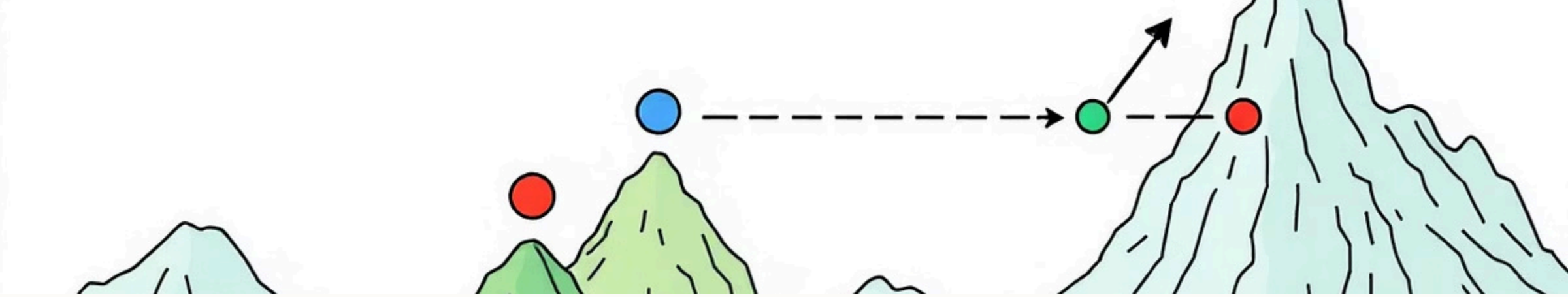


Trabajo Práctico 3: Optimización y Problemas de Satisfacción de Restricciones

Estrategias de búsqueda local, algoritmos evolutivos y problemas de satisfacción de restricciones



1. Algoritmo de Ascensión de Colinas

Mecanismo de Detención

Se detiene cuando no encuentra vecinos con mejor valor que el actual

Problema Principal

Puede quedar atrapado en máximos locales sin alcanzar el máximo global

2. Heurísticas en Problemas de Satisfacción de Restricciones

Mayor Grado Heurístico

Prioriza variables con más conexiones

Mínimos Valores Restantes

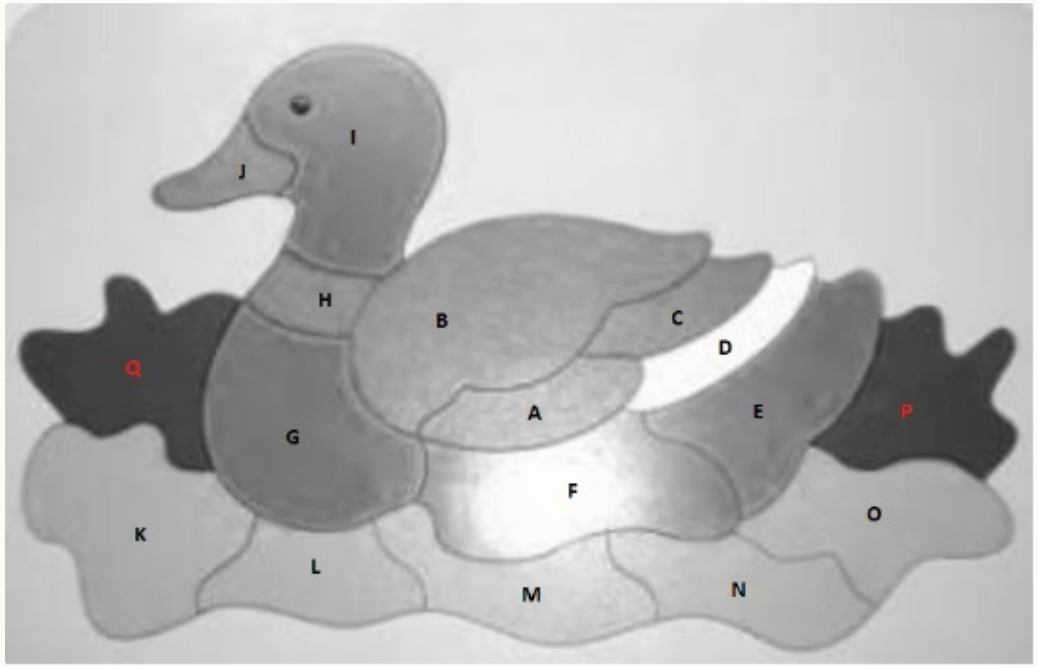
Selecciona variables con menos opciones disponibles

Valor Menos Restringido

Elige valores que mantienen flexibilidad en el resto del problema

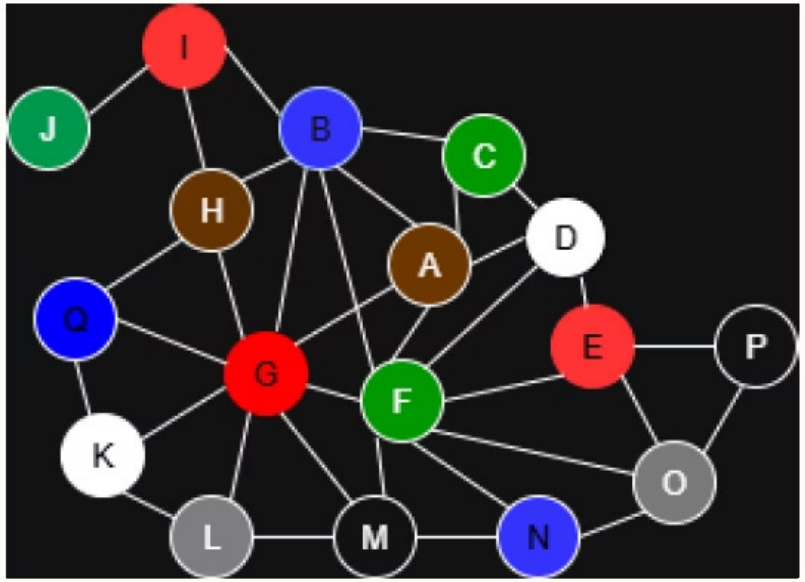
3. Coloración de Rompecabezas

Problema: Colorear piezas con 7 colores distintos sin que las vecinas tengan el mismo color



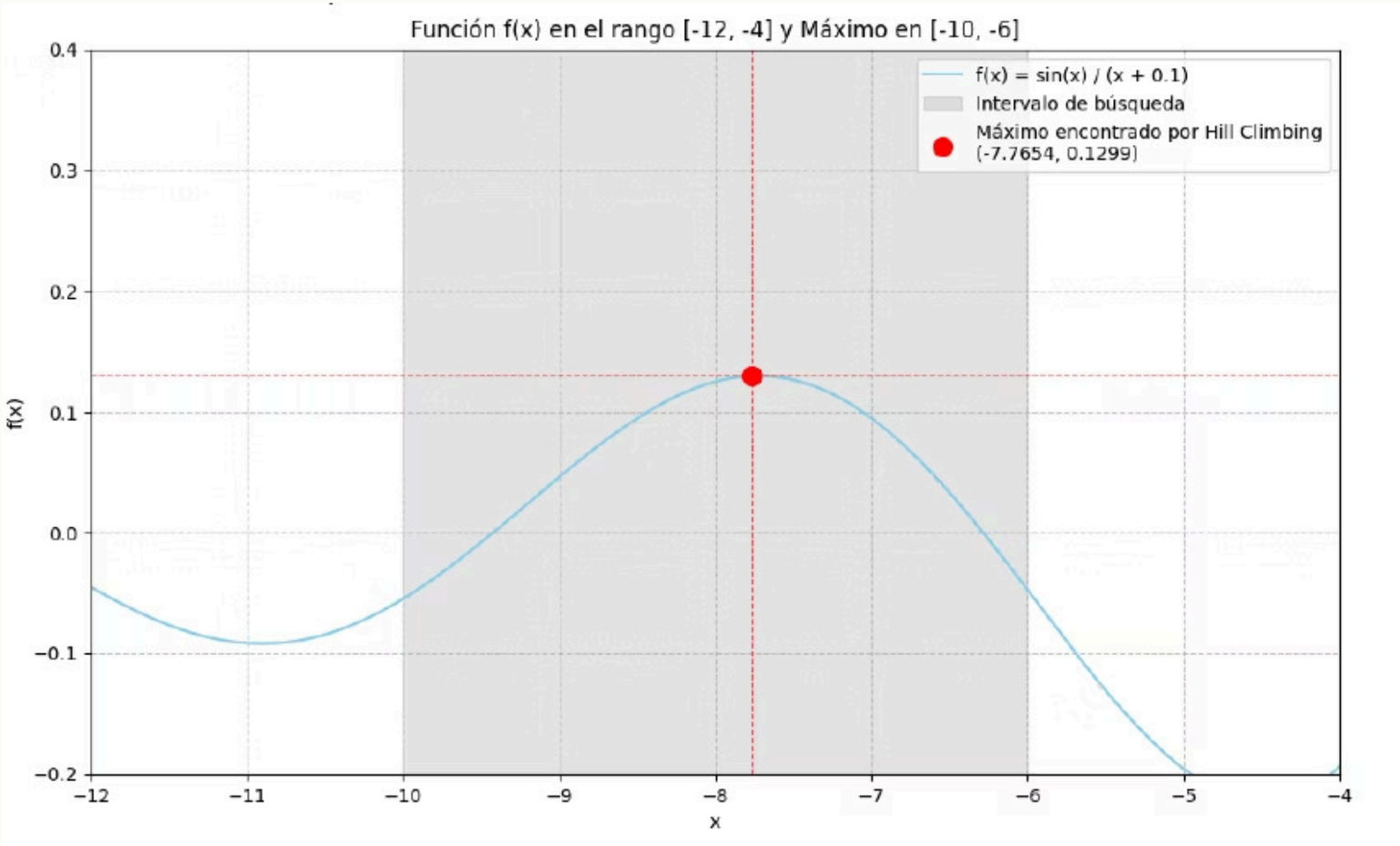
Solución mediante Comprobación hacia Adelante con heurística del Valor más Restringido

Se asignan colores comenzando por las piezas con más restricciones (G, F, B...)



4. Maximización con Hill Climbing

Función a maximizar: f(x) = sin(x)/(x+0.1) en x∈[-10;-6]



Implementación

Algoritmo con múltiples reinicios aleatorios para evitar máximos locales

Resultado

Máximo encontrado: 0.1299 en x = -7.7654

5. Recocido Simulado: Ta-te-ti

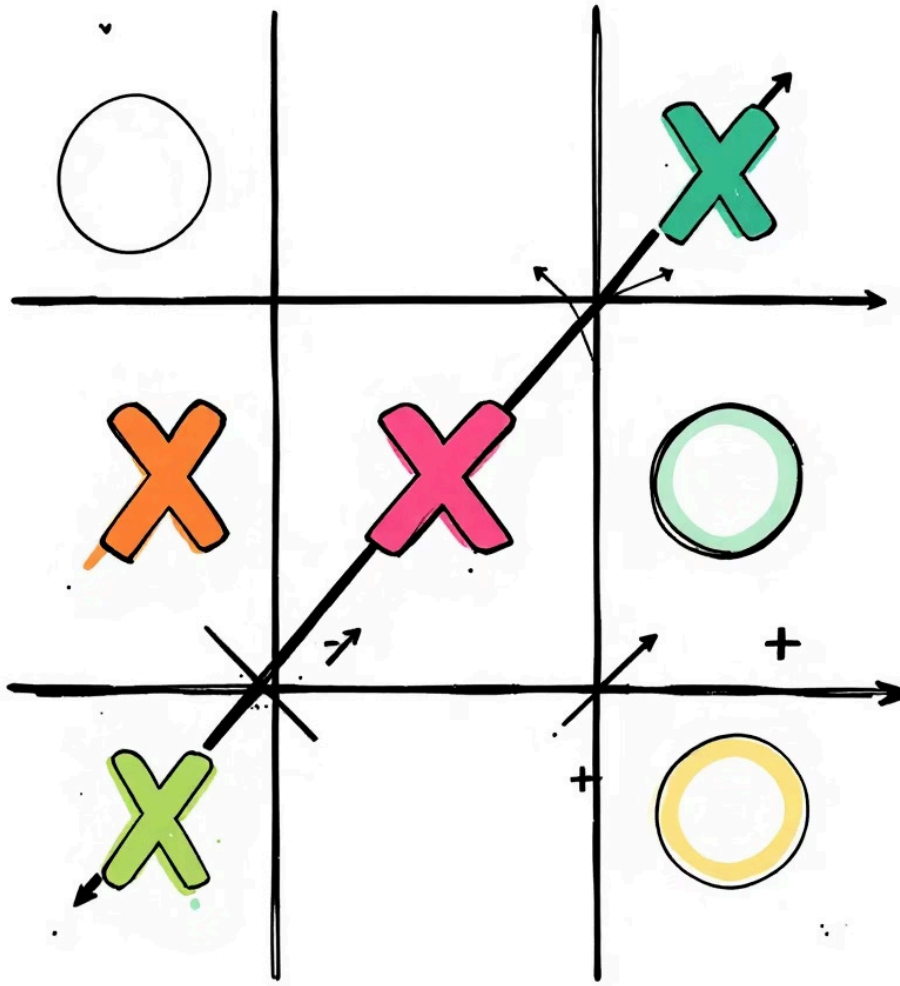
Características

- Algoritmo que juega contra humano
- Evalúa tablero con función de costo
- Temperatura controla aleatoriedad

Efecto de la Temperatura

Alta temperatura: Más exploración, movimientos menos predecibles

Baja temperatura: Más explotación, movimientos más óptimos



6. Algoritmo Genético: Problema de la Grúa

Objetivo: Maximizar valor sin superar 1000kg de peso

Representación

Cromosoma binario: 1 = caja seleccionada, 0 = caja no seleccionada

Población

Conjunto de soluciones válidas (no exceden peso máximo)

Operadores

Selección por ruleta, cruce de un punto, mutación por bit-flip

6.1Datos del Problema de la Grúa

10 Cajas Disponibles

Pesos: 300, 200, 450, 145, 664, 90, 150, 355, 401, 395 kg

Valores: 100, 50, 115, 25, 200, 30, 40, 100, 100, 100 $

Capacidad máxima: 1000 kg

```
    Parametros:
        tamano: Número de individuos en la población.
        cajas: Lista de cajas disponibles.
        inicializar: Si es True, genera individuos aleatorios.
    """
    def __init__(self, tamano: int, cajas: List[Caja], inicializar: bool = True)
        self.individuos = []
        self.cajas = cajas

        if inicializar:
            for _ in range(tamano):
                cromosoma = [np.random.choice([True, False]) for _ in range(len(
                individuo = Individuo(cromosoma, cajas)
                self.individuos.append(individuo)

    def obtener_mejor_individuo(self) -> Individuo:
        return max(self.individuos, key=lambda ind: ind.fitness)

    def tamano(self) -> int:
        return len(self.individuos)
```

In [8]:

```
"""
Definición de la clase AlgoritmoGenetico.
"""
class AlgoritmoGenetico:
    """
    Inicializa el algoritmo genético con los parámetros dados.

    Parametros:
        tamano_poblacion: Número de individuos en la población.
        tasa_mutacion: Probabilidad de mutación para cada gen.
        tasa_cruce: Probabilidad de cruce entre dos individuos.
        cajas: Lista de cajas disponibles.
        elitismo: Si es True, conserva el mejor individuo en cada generación.
    """
    def __init__(self, tamano_poblacion: int, tasa_mutacion: float, tasa_cruce:
                 cajas: List[Caja], elitismo: bool = True):
        self.tamano_poblacion = tamano_poblacion
        self.tasa_mutacion = tasa_mutacion
        self.tasa_cruce = tasa_cruce
        self.cajas = cajas
        self.elitismo = elitismo
        self.historial_fitness = []  # Para almacenar el mejor fitness de cada g

    def ejecutar(self, generaciones: int) -> Individuo:
        """
        Ejecuta el algoritmo genetico para encontrar la mejor solucion.

        Argumentos:
            generaciones (int): Numero de generaciones a evolucionar.

        Retorna:
            Individuo: El mejor individuo encontrado tras todas las generaciones
        """
        # Inicializacion. Crear población inicial aleatoria
        poblacion = Poblacion(self.tamano_poblacion, self.cajas)
        mejor_global = poblacion.obtener_mejor_individuo()
```

Resultados del Algoritmo Genético

300	964	2
Valor Total ($)	**Peso Total (kg)**	**Cajas Seleccionadas**
Máximo valor obtenido	Dentro del límite de 1000kg	Caja 1 (300kg, $100) y Caja 5 (664kg, $200)