CSP

```
Class CSPProblem:
    def __init__(self)
        self.X = [X1, X2, ..., Xn]
        self.D = {variable: [valores] for variable in self.X}
    def restricciones_unarias(X, v)
        # regresa True si v no puede ser valor de X.
        return v not in self.D[X]
```

X: Variables , D: diccionario con valores que pueden tener las variables

$CSP.X = \{X_1, \dots, X_n\}$

$CSP.D = \{X_1 : D_1, \dots, X_n : D_n\}$ donde $D_i = \{V_{i1}, \dots, V_{im}\}$

$CSP.N = \{X_1 : N_1, \dots, X_n : N_n\}$ donde N_i son vecinos de X_i

$CSP.\text{restricción\_binaria}(X_A, V_A, X_B, V_A)$:

variables = $X_1, X_2, \dots, X_n$

$X \in \mathcal{X}$ espacio discreto

$\mathcal{X} = \{X^{(1)}, X^{(2)}, X^{(3)}, \dots\}$

card |X| = cantidad de puntos en el espacio

costo(X) = J(X) es el costo de X

$J: \mathcal{X} \to \mathbb{R}$

Si X es un espacio métrico

$\exists\ d: X \times X \to \mathbb{R}$ tq

$d(x,x) = 0 \quad \forall x \in X$

$d(x,y) = d(y,x) \quad \forall x,y \in X$

$d(x,y)^2 + d(y,z)^2 \geq d(x,z)^2$

Si X es un espacio topológico:

$\exists$ vecino: $X \to \mathcal{P}(X)$ tq

vecinos(x) = $V \subseteq X$ Los vecinos de X

donde $x \subseteq V$

Mínimo global en J

$x \in \mathcal{X}$ es un mínimo global si:

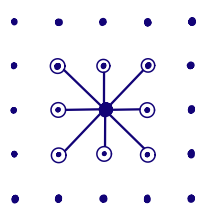
$J(x) \leq J(x') \quad \forall x' \in \mathcal{X}$

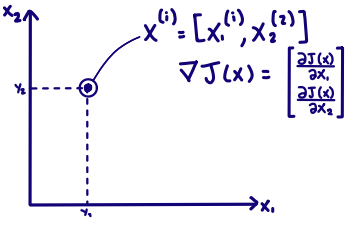
Mínimo local en J

$x \in \mathcal{X}$ es un mínimo local si:

$J(x) \leq J(x')\ .\ \forall x' \in \text{Vecinos}(x)$

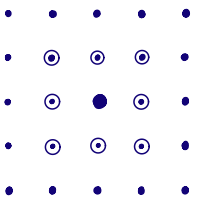
$J(x) = J([x_1, x_2])$



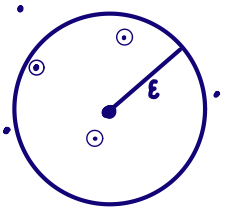


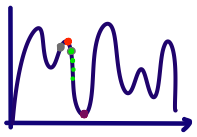
vecinos en un espacio

discreto:

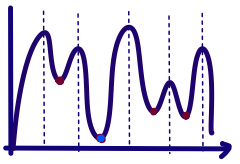


continuo:





1. Punto aleatorio
2. Buscar vecinos
3. Te vas al más bajo
4. Repetir hasta llegar al min



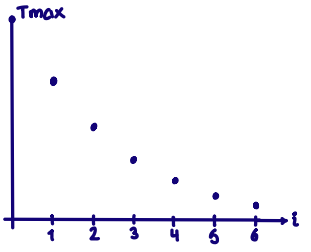
Encontraremos el mínimo de cada región.
- El min local

→ Este algoritmo es muy lento porque hay que reiniciarlo aleatoriamente para el min global

- Aquí entra el temple simulado

calendarización

```
simulated_annealing(pbl, calen, ε)
    x = pbl.estado_aleatorio()
    c = pbl.costo(x)
    for i in range(1, 100_000):
        T = calen(i)
        v = pbl.vecino_aleatorio(x)
        cv = pbl.costo(v)
        if c-cv ≥ 0 and random.random() <= exp(-(c-cv)/T):
            x,c = v, cv
        if T <= ε:
            break
```



Típicamente, Tmax = 10 * num de dimensiones

Calen(i) = Tmax/i = Tmax/ln(i)+1 = Tmax · $e^{-K(i-1)}$

```
class NR(—):
    def __init__(self,N)

def desc_colinas(pbl, max_iter)
    x = pbl.estado aleatorio()
    c = pbl.costo(x)
    for _ in range(max_iter):
        mínimo = True
        for v in pbl.vecinos(x):
            cv = pbl.costo(v)
            if cv < c:
                x,c = v, cv
                minimo = False
        if minimo:
            break
    return x,c
```

```
class PblBúsquedaLocal:
    def estado_aleatorio(self):
        return x∈X en forma aleatoria
    def vecinos(self, x):
        return iterable
            ...
```