PROBLEMAS DE OPTIMIZACIÓN

Inteligencia Artificial

CEIA- FIUBA

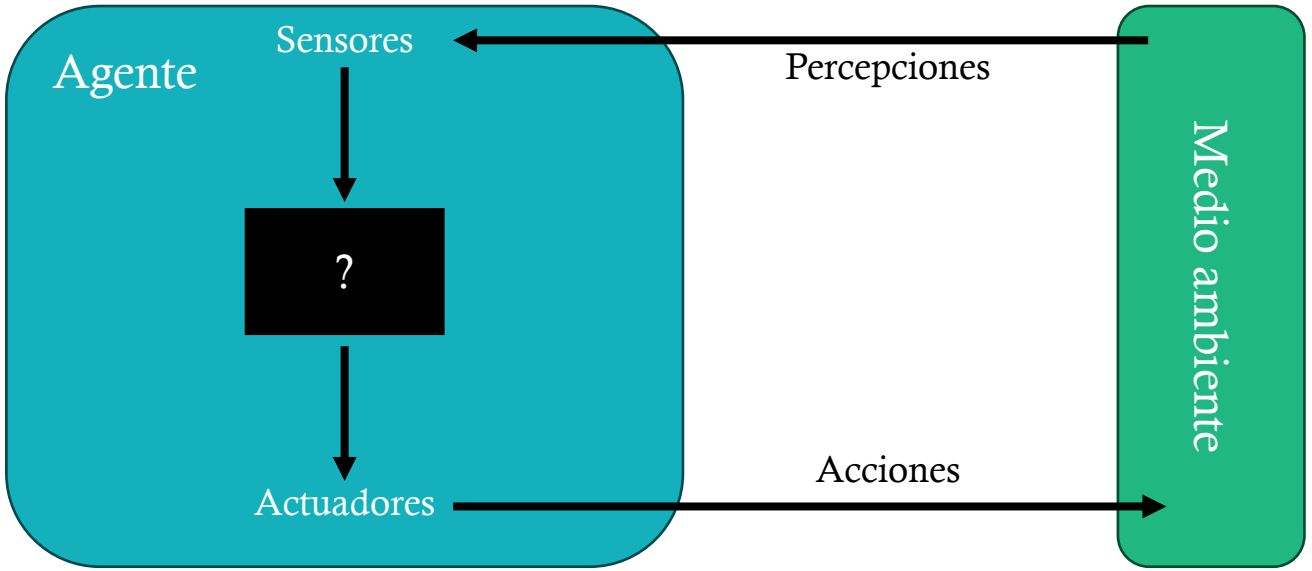
Dr. Ing. Facundo Adrián Lucianna

LO QUE VIMOS LA CLASE ANTERIOR…

AGENTES RACIONALES

Agente

Un agente es cualquier cosa capaz de percibir su **medioambiente** con la ayuda de **sensores** y actuar en ese medio utilizando **actuadores**.



AGENTES RACIONALES

Agente

El término **percepción** se utiliza para indicar que el agente puede recibir entradas en cualquier instante.

La **secuencia de percepciones** de un agente refleja el historial completo de lo que el agente ha recibido.

Una elección de acción de un agente en un momento dado puede depender en su conocimiento incorporado y en la secuencia completa de percepciones hasta ese instante, pero no en cualquier cosa que no haya percibido.

AGENTES RACIONALES

Racionalidad

La racionalidad en un momento dado depende de cuatro factores:

- La medida de rendimiento que define el criterio de éxito.
- El conocimiento previo del agente sobre el entorno..
- Las acciones que el agente puede llevar a cabo.
- La secuencia de percepciones del agente hasta este momento.

Definición de racionalidad

En cada posible secuencia de percepciones, un agente racional deberá seleccionar aquella acción que supuestamente maximice su medida de rendimiento, basándose en las evidencias aportadas por la secuencia de percepciones y en el conocimiento que el agente mantiene almacenado.

AGENTES RACIONALES

Especificación del entorno de trabajo

En el ejemplo de racionalidad de una agente aspiradora, hubo que especificar las medidas de rendimiento, el entorno, y los actuadores y sensores del agente.

Todo ello forma lo que se llama el entorno de trabajo, cuya denominación es **PEAS**:

- **Performance**
- **Environment**
- **Actuators**
- **Sensors**

AGENTES RACIONALES

Propiedades del entorno de trabajo

Veamos algunas dimensiones que podemos categorizar a los entornos:

- Totalmente observable vs. parcialmente observable
- Deterministas vs. Estocástico
- Episódico vs. Secuencial
- Estático vs. Dinámico
- Discreto vs. Continuo
- Agente individual vs. Multiagente (competitivo o cooperativo)

RESOLUCIÓN DE PROBLEMAS MEDIANTE BÚSQUEDA

Agentes de resolución de problemas

Cuando la acción correcta a tomar no es inmediatamente obvia, un agente puede necesitar planificar con anticipación: considerar una secuencia de acciones que formen un camino hacia un estado objetivo. A dicho agente se le llama **agente de resolución de problemas** y el proceso computacional que lleva a cabo se llama **búsqueda**.

Para estos métodos de búsquedas, se considera sólo los entornos más simples: *episódico, de agente único, totalmente observable, determinista, estático, discreto y conocido*.

Dado estas condiciones, el agente puede llevar un proceso de 4 fases:

- **Formulación de objetivo:** El agente adopta el objetivo basado en la situación actual y la medida de rendimiento del agente.
- **Formulación del problema:** El agente diseña una descripción de los estados y acciones necesarias para alcanzar el objetivo: un modelo abstracto de la parte relevante del entorno.
- **Búsqueda:** Antes de realizar cualquier acción en el mundo real, el agente simula secuencias de acciones en su modelo, buscando hasta encontrar una secuencia de acciones que alcance el objetivo. Esta secuencia se llama solución.
- **Ejecución:** El agente ahora puede ejecutar las acciones de la solución, de a un paso por vez.

ALGORITMOS DE BÚSQUEDA

- **Algoritmos de búsqueda no informada:**
  - Búsqueda primero en anchura
  - Búsqueda de costo uniforme
  - Búsqueda primero en profundidad
  - Búsqueda de profundidad limitada
  - Búsqueda de profundidad limitada con profundidad iterativa
- **Algoritmos de búsqueda informada:**
  - Búsqueda voraz (greedy) primero el mejor
  - Búsqueda A*

ALGORITMOS DE BÚSQUEDA LOCAL

ALGORITMOS DE BÚSQUEDA LOCAL

Los algoritmos de búsqueda que vimos la clase anterior se diseñan para explorar sistemáticamente espacios de búsqueda. Esta forma sistemática se alcanza manteniendo uno o más caminos en memoria y registrando qué alternativas se han explorado en cada punto a lo largo del camino y cuáles no.

Cuando se encuentra un objetivo, **el camino** a ese objetivo también constituye una **solución** al problema.

Pero hay problemas en donde **no** nos importa el camino, sino que importa la configuración final. Por ejemplo, un algoritmo que resuelva Sudokus.

3	7	4	5	6	1	9	2	8
1	8	5	4	2	9	7	6	3
9	6	2	3	7	8	4	1	5
8	2	7	6	1	3	5	4	9
6	4	9	2	5	7	8	3	1
5	3	1	9	8	4	6	7	2
4	9	6	8	3	2	1	5	7
2	1	8	7	4	5	3	9	6
7	5	3	1	9	6	2	8	4

ALGORITMOS DE BÚSQUEDA LOCAL

Si no importa el camino al objetivo, podemos considerar una clase diferente de algoritmos que no se preocupen en absoluto de los caminos.

Los algoritmos de búsqueda local funcionan con un solo estado actual y generalmente se mueve sólo a los *vecinos* del estado. Estos algoritmos tienen dos ventajas:

1. Usan poca memoria
2. Pueden encontrar soluciones razonables en espacios de estado grandes o infinitos (continuos).

ALGORITMOS DE BÚSQUEDA LOCAL

Problemas de optimización

Los algoritmos de búsqueda local son útiles para resolver **problemas de optimización** puros, en los cuales el objetivo es encontrar el mejor estado según una función objetivo.

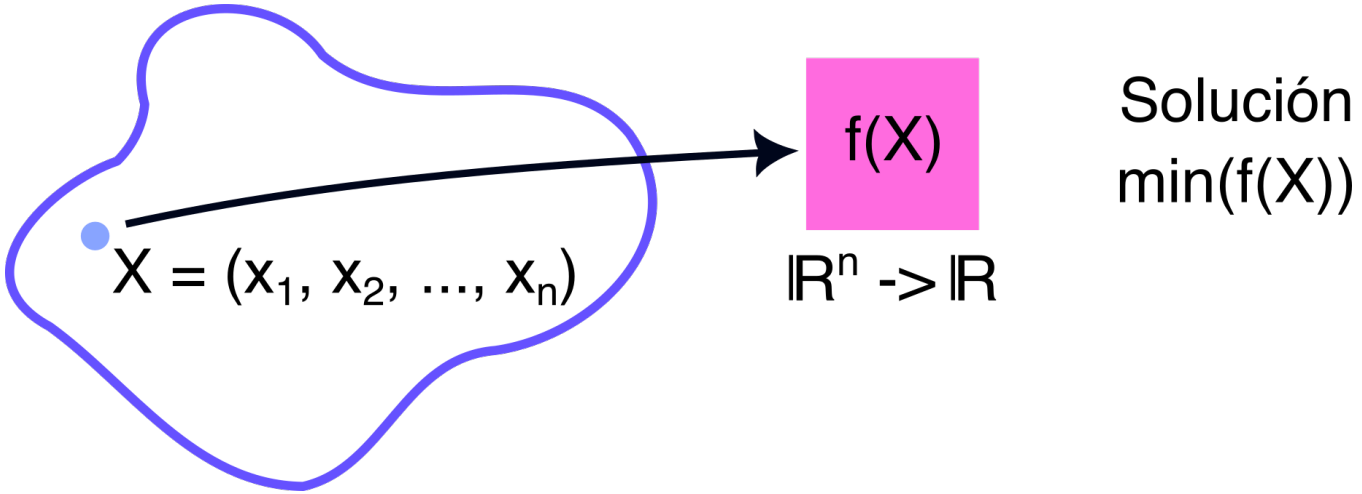
Un problema de optimización consiste en minimizar o maximizar el valor de una variable. En otras palabras, se trata de calcular o determinar el valor mínimo o el valor máximo de una función.

Los problemas de optimización se pueden dividir en dos categorías:

- Un problema de optimización con variables discretas se conoce como *optimización discreta*, en la que un objeto como un número entero, una permutación o un grafo se debe encontrar en un conjunto contable.
- Un problema con variables continuas se conoce como *optimización continua*, en la que se debe encontrar un valor óptimo de una función continua.

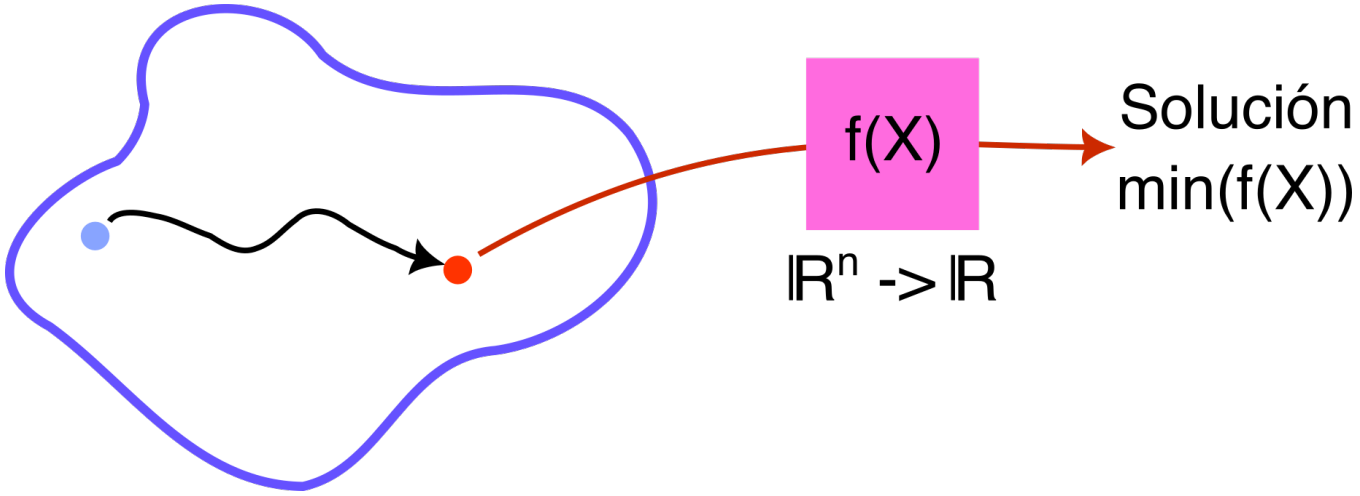
ALGORITMOS DE BÚSQUEDA LOCAL

Problemas de optimización



ALGORITMOS DE BÚSQUEDA LOCAL

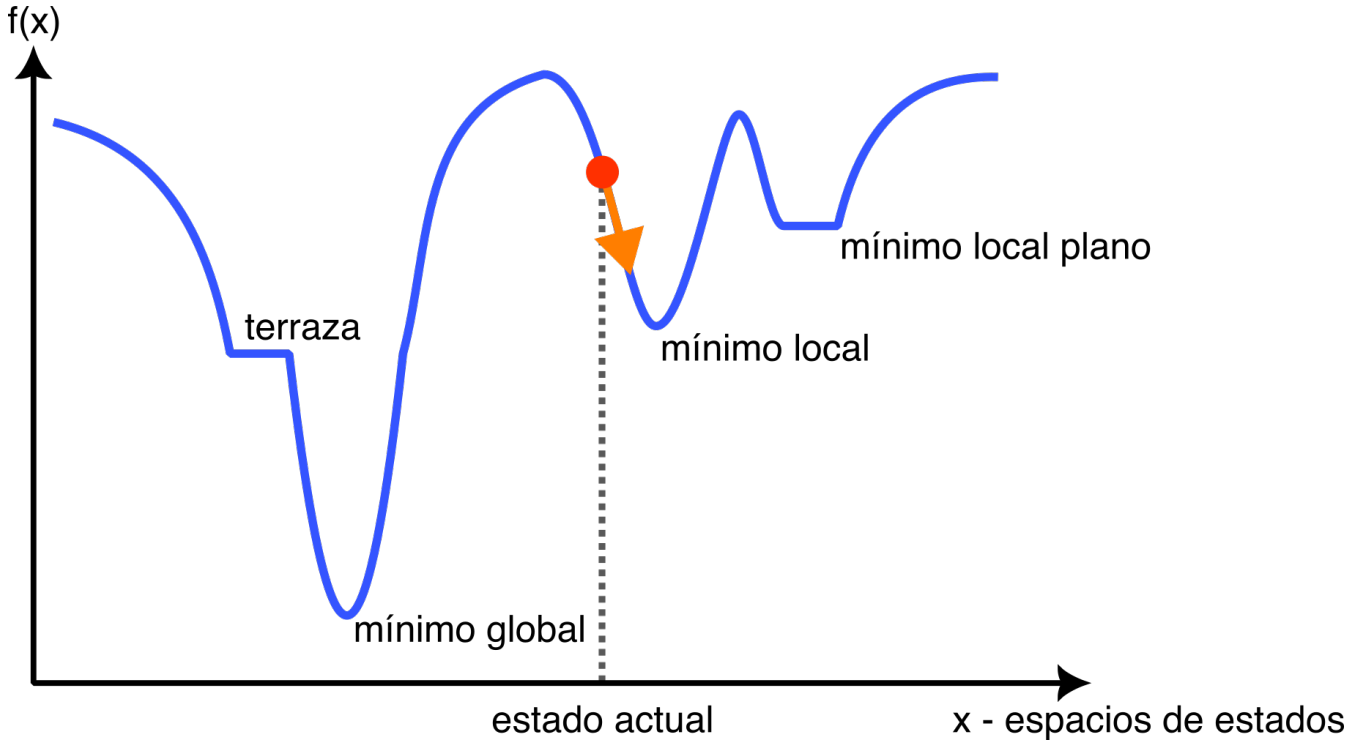
Problemas de optimización



ALGORITMOS DE BÚSQUEDA LOCAL

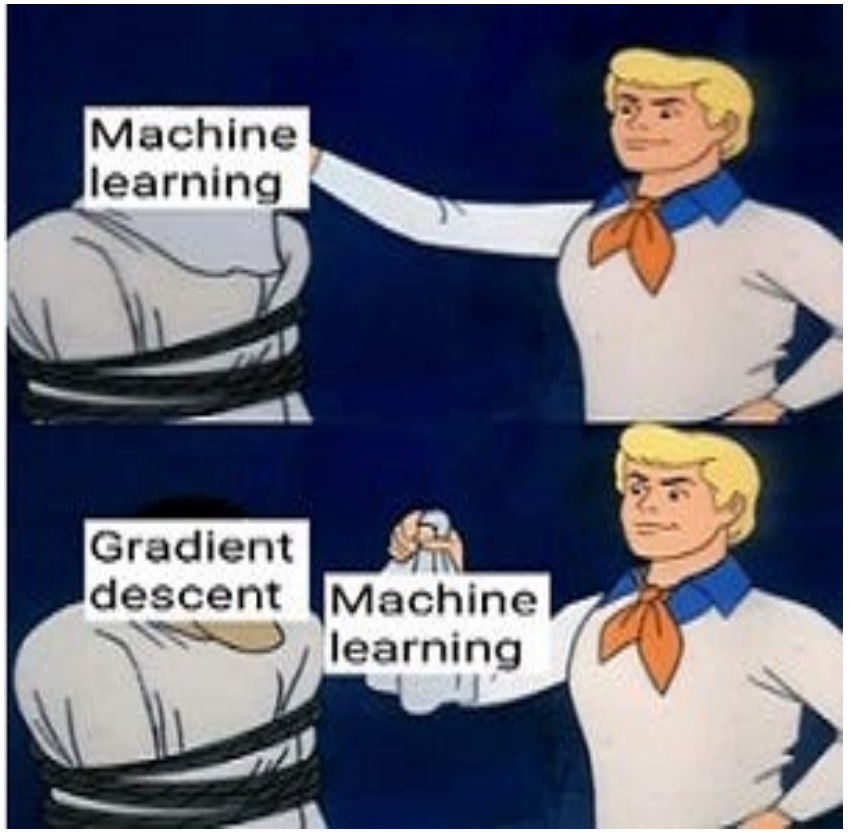
Problemas de optimización

Pasemos a un caso de una sola variable así podemos observar:



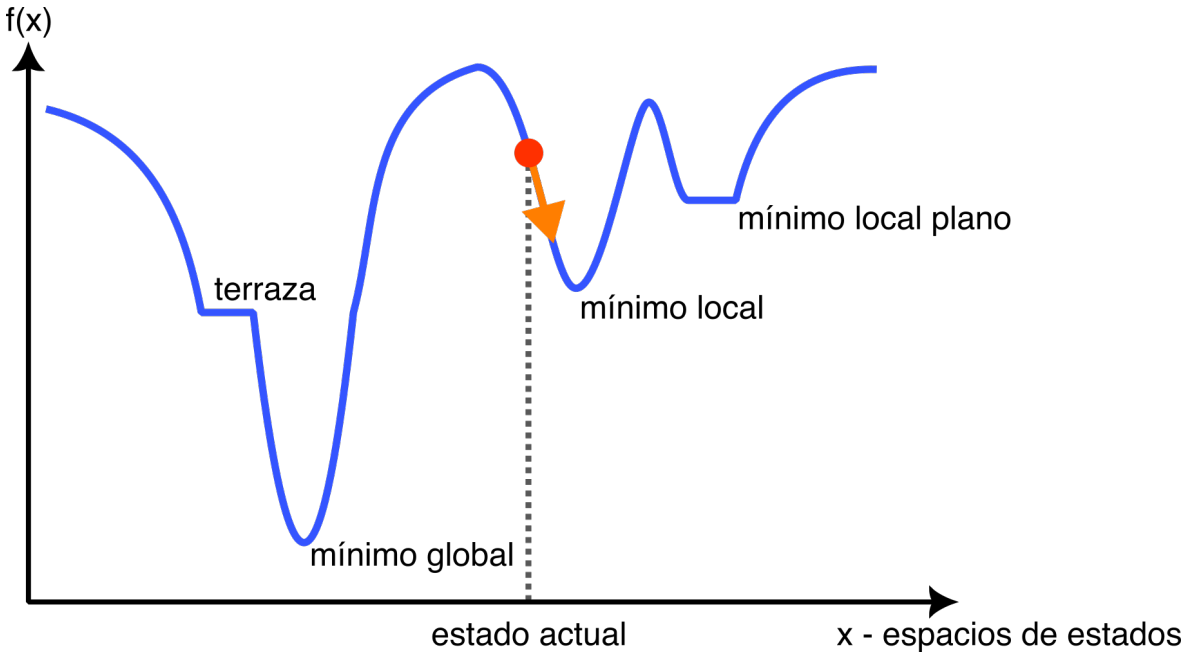
GRADIENTE DESCENDIENTE O ASCENDENTE

GRADIENTE DESCENDIENTE O ASCENDENTE



GRADIENTE DESCENDIENTE O ASCENDENTE

Este algoritmo que continuamente se mueve en dirección de mayor valor decreciente o creciente. Termina la búsqueda en donde ningún vecino está más bajo. Este algoritmo no miras más allá de lo que tiene de los vecinos.




```python
def clim_hill(x_initial, fun_opt):
  x = x_initial
  f_x = fun_opt(x_initial)

  while 1:
    # Obtenemos vecino
    x_neib = obtain_neib(x)
    # Si f del vecino es menor
    if fun_opt(x) <= f_x:
      x = x_neib
      f_x = fun_opt(x)
    else:
      # Si llegamos a un minimo, termina
      return x
```

GRADIENTE DESCENDIENTE O ASCENDENTE

Esta búsqueda se le llama búsqueda local voraz (greedy), porque avanza a un estado vecino sin pensar en donde ir después.

Avanza muy rápido hacia una solución, pero es fácil llegar a un estado no ideal. Se atasca por los siguientes motivos:

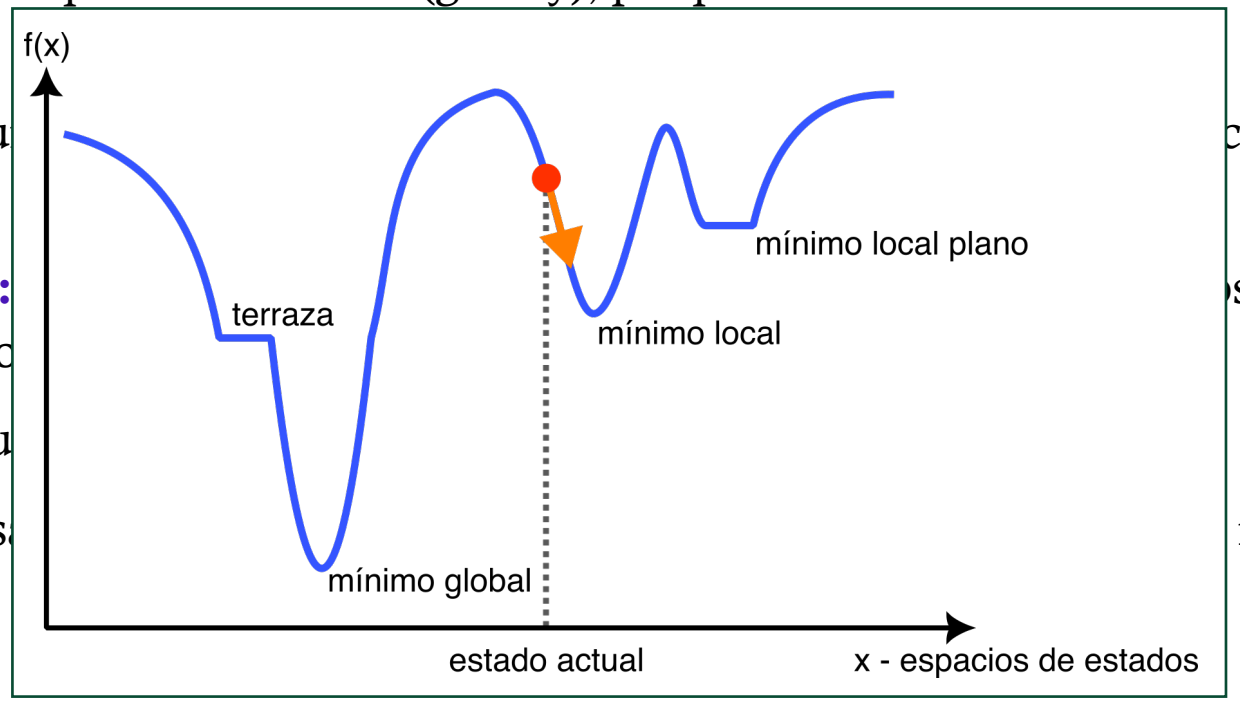
- **Mínimo (máximo) local:** Un mínimo local es un valle que es más bajo que sus estados vecinos, pero estás más arriba que el mínimo global.
- **Meseta:** Una meseta es un área donde la función de evaluación es plana.
- **Crestas:** Las crestas causan una secuencia de mínimos locales que hace muy difícil la navegación.

GRADIENTE DESCENDIENTE O ASCENDENTE

Esta búsqueda se le llama búsqueda local voraz (greedy), porque avanza a un estado vecino sin pensar en donde ir después.

Avanza muy rápido hacia u ca por los siguientes motivos:

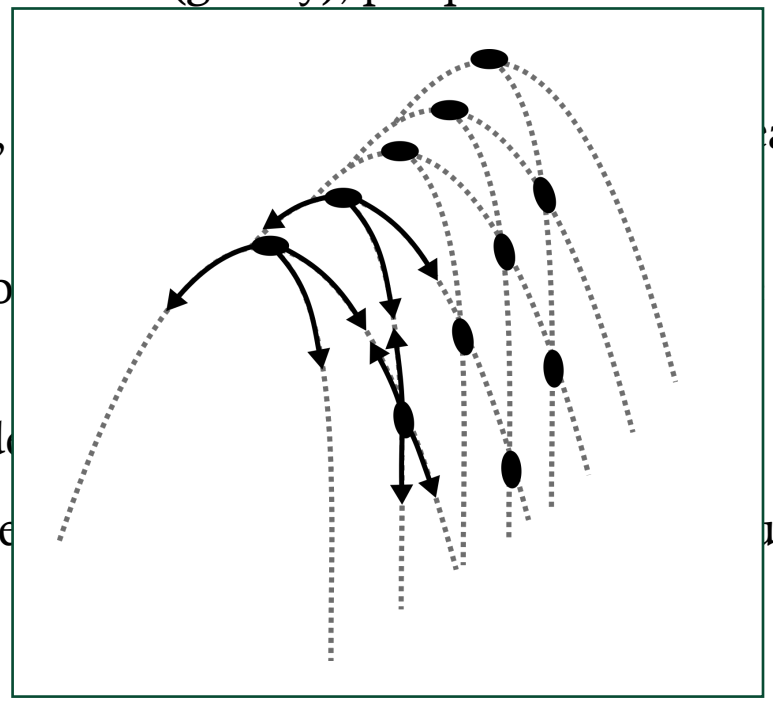
- **Mínimo (máximo) local:** s vecinos, pero estás más arriba que el mínimo
- **Meseta:** Una meseta es u
- **Crestas:** Las crestas caus navegación.



GRADIENTE DESCENDIENTE O ASCENDENTE

Esta búsqueda se le llama búsqueda local voraz (greedy), porque avanza a un estado vecino sin pensar en donde ir después.

Avanza muy rápido hacia una solución, al. Se atasca por los siguientes motivos:



- **Mínimo (máximo) local:** Un mínimo sus estados vecinos, pero estás más arriba que el mínimo global.
- **Meseta:** Una meseta es un área dond
- **Crestas:** Las crestas causan una secue y difícil la navegación.

GRADIENTE DESCENDIENTE O ASCENDENTE

Esta búsqueda se le llama búsqueda local voraz (greedy), porque avanza a un estado vecino sin pensar en donde ir después.

Avanza muy rápido hacia una solución, pero es fácil llegar a un estado no ideal. Se atasca por los siguientes motivos:

- **Mínimo (máximo) local:** Un mínimo local es un valle que es más bajo que sus estados vecinos, pero estás más arriba que el mínimo global.
- **Meseta:** Una meseta es un área donde la función de evaluación es plana.
- **Crestas:** Las crestas causan una secuencia de mínimos locales que hace muy difícil la navegación.

Para lograr salir de mesetas, se puede permitir movimientos laterales (es decir que no cambia la función) pero para evitar entrar en un bucle infinito, se puede poner un límite de movimientos laterales consecutivos.

GRADIENTE DESCENDIENTE O ASCENDENTE

Variantes de este algoritmo:

- **Ascenso o descenso estocástico:** Se escoge aleatoriamente la dirección de entre los movimientos ascendentes posibles. Converge más lento, pero en ciertos casos encuentra mejor solución.
- **Ascenso o descenso estocástico de primera opción:** Posee una planificación el cual genera sucesores de forma estocástica al azar hasta que se genera una que es mejor que el estado actual
- **Ascenso o descenso de reinicio aleatorio:** Esto conduce a una serie de búsquedas en ascensión de colinas desde estados iniciales generados aleatoriamente, parándose cuando se encuentra un objetivo.

GRADIENTE DESCENDIENTE O ASCENDENTE

Veamos la aplicación de gradiente descendiente para un problema de Sudoku…

3		4	5	6		9		
1	8	5			9	7		
				7	8	4	1	5
	2			1			4	9
	4	9		5				
		1	9	8		6	7	
4	9			3				7
	1	8	7	4	5			6
							8	

3	7	4	5	6	1	9	2	8
1	8	5	4	2	9	7	6	3
9	6	2	3	7	8	4	1	5
8	2	7	6	1	3	5	4	9
6	4	9	2	5	7	8	3	1
5	3	1	9	8	4	6	7	2
4	9	6	8	3	2	1	5	7
2	1	8	7	4	5	3	9	6
7	5	3	1	9	6	2	8	4

SIMULATED ANNEALING

SIMULATED ANNEALING

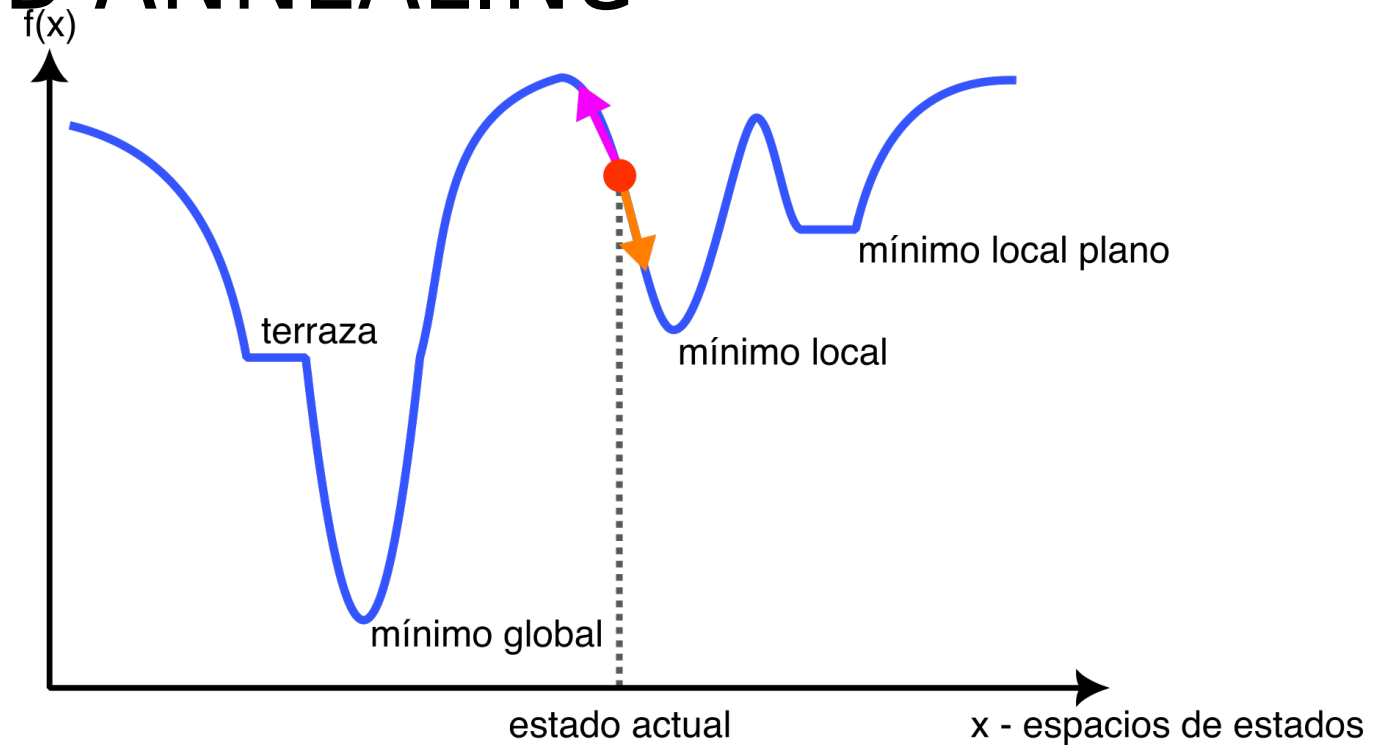
Un algoritmo de gradiente descendiente o ascendente que nunca hace movimientos de ascenso hacia estados de coste más alto garantiza ser **incompleto**, porque puede estancarse en un mínimo local.

En contraste, un camino puramente aleatorio, es **completo**, pero sumamente ineficaz.

Por lo que si usamos un método que use un poco de cada uno podemos encontrar el **equilibrio** entre completitud y eficacia. **Simulated annealing** ese algoritmo.

En metalurgia, el **recocido (annealing)** es el proceso para endurecer metales calentándolos a una temperatura alta y luego gradualmente enfriarlos, asi permite al material fundirse en un estado cristalino de baja energía.

SIMULATED ANNEALING



La idea aquí es que en general se vaya a un estado de menor energía, pero por azar a veces pueda ir en **dirección contraria**, buscando de esa forma salir de un mínimo local. La idea es sacudir para que salga del mínimo local pero que no se nos vaya tan lejos que no podamos llegar al mínimo global

SIMULATED ANNEALING

La idea para lograr esta sacudida al azar es usar un nuevo parámetro que llamamos temperatura. Cuando más temperatura haya más probable de que podamos movernos en dirección contraria, cuando baja la temperatura, baja esta probabilidad. Si la temperatura es cero estamos en un algoritmo de descenso de colina puro.

Este algoritmo arranca con mucha temperatura y a medida que avanza, se va enfriando, de igual forma que el metal.

Se puede demostrar que, si se disminuye la temperatura bastante despacio, el algoritmo encontrará un óptimo global con probabilidad cerca de uno.

```
def sim_anneal(x_initial, fun_opt, temp_initial, rate_cooling):
  x = x_initial
  f_x = fun_opt(x_initial)
  T = temp_initial

  while T > 0.00001:
    # Obtenemos vecino
    x_neib = obtain_neib(x)
    deltaE = f_x - fun_opt(x)
    # Si f del vecino es menor, se acepta
    if deltaE > 0:
      x = x_neib
      f_x = fun_opt(x)
    else:
      # Si no, veamos si por distribución de
      # Boltzmann se acepta
      num_ale = random() # Numero entre 0 y 1
      # Si por azar es menor a exp(-deltaE/T)
      if num_ale < exp(-deltaE/T):
        # Aceptamos el cambio a pesar que
        # aumente el costo.
        x = x_neib
        f_x = fun_opt(x)
    # Enfriamos
    T *= rate_cooling

  return x
```

SIMULATED ANNEALING

Para determinar si se hace el cambio, se hace uso de la distribución de Boltzmann, el cual es una distribución de probabilidad de partículas en un sistema a través de varios estados posibles:

$$F \propto e^{-\frac{\Delta E}{kT}}$$

Esta distribución establece que cuando más grande es la diferencia de energía (o en nuestro caso la diferencia de la función de costo) menos probable es que se elija, pero si la **temperatura es alta, esta probabilidad es mayor y es más fácil que de todas formas se elija.**

```python
def sim_anneal(x_initial, fun_opt, temp_initial, rate_cooling):
  x = x_initial
  f_x = fun_opt(x_initial)
  T = temp_initial

  while T > 0.00001:
    # Obtenemos vecino
    x_neib = obtain_neib(x)
    deltaE = f_x - fun_opt(x)
    # Si f del vecino es menor, se acepta
    if deltaE > 0:
      x = x_neib
      f_x = fun_opt(x)
    else:
      # Si no, veamos si por distribución de
      # Boltzmann se acepta
      num_ale = random() # Numero entre 0 y 1
      # Si por azar es menor a exp(-deltaE/T)
      if num_ale < exp(-deltaE/T):
        # Aceptamos el cambio a pesar que
        # aumente el costo.
        x = x_neib
        f_x = fun_opt(x)
    # Enfriamos
    T *= rate_cooling

  return x
```

SIMULATED ANNEALING

Veamos la aplicación de simulated annealing para un problema de Sudoku…

3		4	5	6		9		
1	8	5			9	7		
				7	8	4	1	5
	2			1			4	9
	4	9		5				
		1	9	8		6	7	
4	9			3				7
	1	8	7	4	5			6
							8	

3	7	4	5	6	1	9	2	8
1	8	5	4	2	9	7	6	3
9	6	2	3	7	8	4	1	5
8	2	7	6	1	3	5	4	9
6	4	9	2	5	7	8	3	1
5	3	1	9	8	4	6	7	2
4	9	6	8	3	2	1	5	7
2	1	8	7	4	5	3	9	6
7	5	3	1	9	6	2	8	4

BÚSQUEDA LOCAL BEAM

BÚSQUEDA LOCAL BEAM

Buscar desde un estado inicial únicamente es una medida un poco extrema de ahorro de memoria.

En cambio, **búsqueda local beam** guarda la información de k estados y sobre ellos realiza la búsqueda independientemente. Estos se inician al azar.

En cada paso se generan sucesores de los k estados. Si alguno cumple el objetivo, se termina, en cambio si no, se seleccionan los k mejores sucesores de la lista.

A simple vista pareciera lo mismo de correr k veces a gradiente descendiente o ascendente, pero a diferencia de esto es que entre los procesos de búsquedas hay pase de información:

Si un estado genera varios sucesores buenos y los otros k-1 estados generan sucesores malos, entonces el efecto es que el primer estado abandona la búsqueda de los otros y se queda con los sucesores del primer estado.

El algoritmo rápidamente abandona las búsquedas infructuosas y mueve sus recursos a donde se hace la mayor parte del progreso.

BÚSQUEDA LOCAL BEAM

El principal **inconveniente** de este algoritmo es que puede sufrir carencia de diversidad de estados, enfocándose en una sola parte muy limitada.

Una variante es la **búsqueda de haz estocástica**, en el cual en vez de elegir a los k mejores sucesores, se eligen aleatoriamente, con una función de temperatura similar a la de *simulated annealing*.

Esta búsqueda muestra un parecido con el proceso de selección natural, por lo cual los *sucesores* (descendientes) de un *estado* (organismo) pueblan la siguiente generación según su *valor* (idoneidad o salud).

BÚSQUEDA LOCAL BEAM

Veamos la aplicación de búsqueda Local Beam para un problema de Sudoku…

3		4	5	6		9		
1	8	5			9	7		
				7	8	4	1	5
	2			1			4	9
	4	9		5				
		1	9	8		6	7	
4	9			3				7
	1	8	7	4	5			6
							8	

3	7	4	5	6	1	9	2	8
1	8	5	4	2	9	7	6	3
9	6	2	3	7	8	4	1	5
8	2	7	6	1	3	5	4	9
6	4	9	2	5	7	8	3	1
5	3	1	9	8	4	6	7	2
4	9	6	8	3	2	1	5	7
2	1	8	7	4	5	3	9	6
7	5	3	1	9	6	2	8	4

ALGORITMOS GENÉTICOS

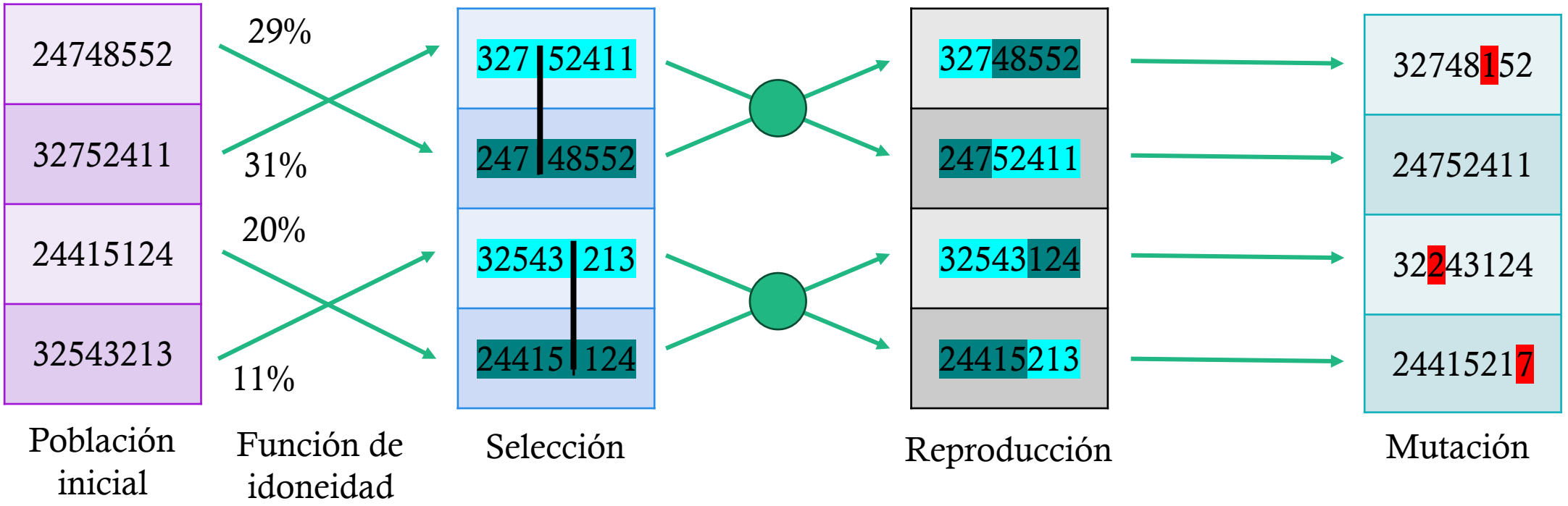
ALGORITMOS GENÉTICOS

Un **algoritmo genético** es una variante de la búsqueda local beam estocátisca en la que los estados sucesores se generan combinando dos estados padres (reproducción), más que modificar un solo estado (reproducción asexual).

La analogía con la selección natural es la misma que anteriormente.

Los algoritmos genéticos comienzan con un conjunto k de estados aleatorios, llamados **población**. Cada estado, o **individuo**, está representado como una cadena de caracteres sobre un **alfabeto finito** que representa el código genético del estado.

ALGORITMOS GENÉTICOS



ALGORITMOS GENÉTICOS

Como en la búsqueda beam local estocástica, los algoritmos genéticos combinan una tendencia ascendente con exploración aleatoria y cambian la información entre los hilos paralelos de búsqueda.

La ventaja del algoritmo genético viene de la operación de cruce para combinar bloques grandes de letras que han evolucionado independientemente para así realizar funciones útiles, de modo que se aumente el nivel de granularidad en el que funciona la búsqueda.

```
def gen_alg(poblation, fun_ido):
  new_poblation = []

  while 1:
    for i in range(len(poblation)):
      x = sel_aleatory(poblation, fun_ido)
      y = sel_aleatory(poblation, fun_ido)
      child = reproduction(x, y)
      dice = random()
      if dice < 0.01:
        child = mutate(child)
      new_poblation.append(child)
    poblation = new_poblation

    for z in poblation:
      if z is solution:
        return z

def reproduction(x, y):
  n = len(x)
  c = random_int(0, n)
  return x[:c] + y[c:]
```

ALGORITMOS GENÉTICOS

Veamos la aplicación de algoritmos genéticos para un problema de Sudoku…

3		4	5	6		9		
1	8	5			9	7		
				7	8	4	1	5
	2			1			4	9
	4	9		5				
		1	9	8		6	7	
4	9			3				7
	1	8	7	4	5			6
							8	

3	7	4	5	6	1	9	2	8
1	8	5	4	2	9	7	6	3
9	6	2	3	7	8	4	1	5
8	2	7	6	1	3	5	4	9
6	4	9	2	5	7	8	3	1
5	3	1	9	8	4	6	7	2
4	9	6	8	3	2	1	5	7
2	1	8	7	4	5	3	9	6
7	5	3	1	9	6	2	8	4

BÚSQUEDA EN ESPACIOS CONTINUOS

BÚSQUEDA EN ESPACIOS CONTINUOS

Los algoritmos que vimos hasta ahora no se pueden aplicar en casos de espacio continuos ya que los sucesores que nos presentan son infinitos.

La literatura de espacio de búsqueda continuas es enorme y es tan vieja como la época de Newton y Liebniz (siglo XVII).

BÚSQUEDA EN ESPACIOS CONTINUOS

Pensemos un ejemplo, se quiere colocar dos aeropuertos nuevos en Argentina, de tal forma que la distancia al cuadrado de cada gran ciudad de Argentina (Buenos Aires, La Plata, Córdoba y Rosario) con respecto a los aeropuertos sea mínima.

Entonces el espacio de estado está definido por 4 coordenadas: (x_1, y_1), (x_2, y_2) correspondiente a la ubicación de los aeropuertos.

Es un espacio cuatro-dimensional o que los estados están definidos por 4 variables. Moverse sobre este espacio se corresponde a moverse a movimientos de los aeropuertos.

La función objetivo $f(x_1, y_1, x_2, y_2)$ es relativamente fácil de calcularla usando la suma de la distancia euclidiana con cada ciudad.

BÚSQUEDA EN ESPACIOS CONTINUOS

Una forma de resolver esto es **discretizar el espacio**.

Por ejemplo, podemos mover de a pequeños pasos a los aeropuertos en dirección x o y, y en una cantidad fija.

Con 4 variables, nos da 8 sucesores para cada estado. De ahí aplicamos cualquier algoritmo de búsqueda local.

BÚSQUEDA EN ESPACIOS CONTINUOS

Hay métodos de usar el gradiente para encontrar un máximo o mínimo:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}\right)$$

En algunos casos podemos resolver la ecuación diferencial:

$$\nabla f = 0$$

Para el ejemplo si tuviéramos un solo aeropuerto, sería la media aritmética de la ubicación de las ciudades.

BÚSQUEDA EN ESPACIOS CONTINUOS

En muchos casos esto no puede resolverse directamente. Pero podemos hacer uso del algoritmo de gradiente, haciendo un cambio local

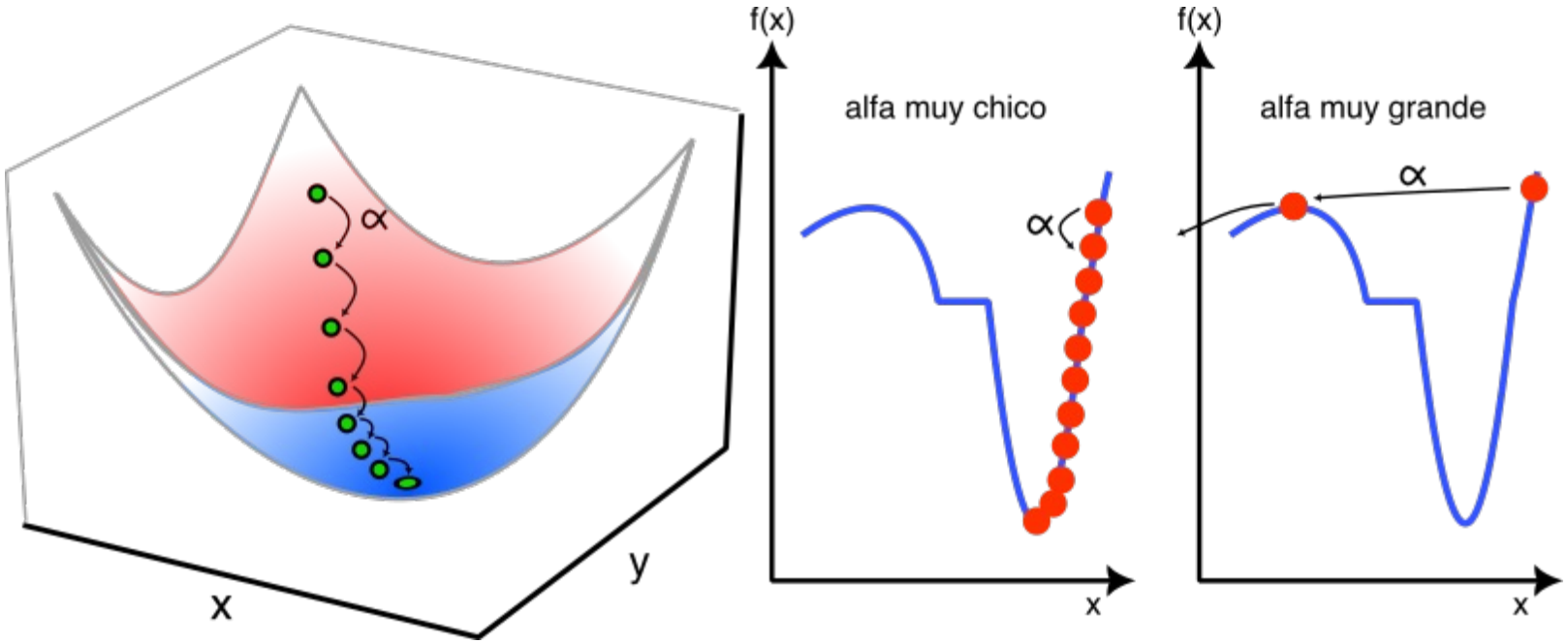
$$X_{new} = X - \alpha \nabla f(X)$$

En donde alfa es una constante pequeña, que en redes neuronales se llama **constante de aprendizaje.**

Hay casos que no tenemos la función objetivo de forma diferenciable. En estos casos se usa el gradiente empírico evaluando pequeño incrementos y decrementos de cada coordenada.

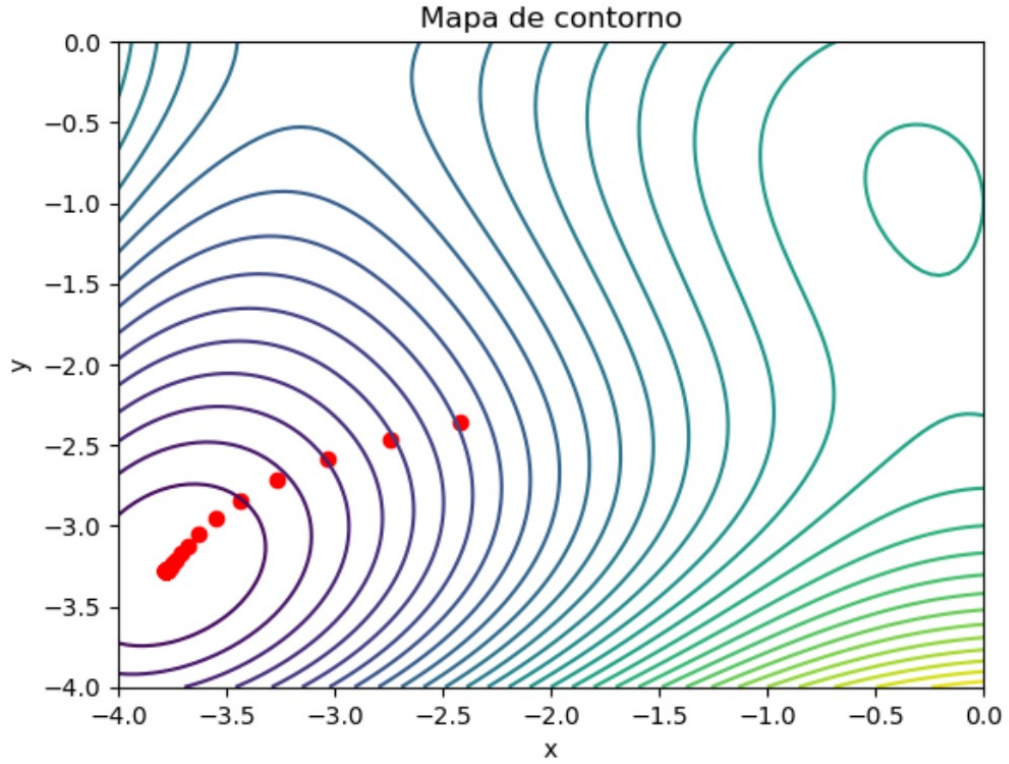
BÚSQUEDA EN ESPACIOS CONTINUOS

La elección de alfa es sumamente importante, porque si alfa es muy pequeña, necesitamos demasiados pasos, en cambio sí es muy grande nunca puede converger.



BÚSQUEDA EN ESPACIOS CONTINUOS

Veamos una aplicación…



LIBRERÍAS DE OPTIMIZACIÓN

Una librería que nos puede ayudar a aplicar estas optimizaciones o más complejas, es OR-Tools de Google.

Es un paquete de software de código abierto para optimización, diseñado para abordar los problemas difíciles en rutas de vehículos, flujos, programación lineal y entera, y programación de restricciones.

Se puede modelar un problema en el lenguaje de programación de elección (C++, Python, Java, etc.), y luego se puede utilizar una media docena de solucionadores para resolverlo, tanto comerciales como de código abierto.