401 ALGORITMOS DE BÚSQUEDAS

September 25, 2025

1 Busquedas por fuerza bruta

1.1 Fundamentos téoricos

El algoritmo de búsqueda por fuerza bruta es una técnica de optimización que examina sistemáticamente todas las soluciones posibles dentro de un espacio de búsqueda definido. A través de una evaluación exhaustiva, selecciona la solución que optimiza una función objetivo preestablecida.

A diferencia de métodos más avanzados que explotan la estructura del problema la fuerza bruta no hace suposiciones sobre la topología del espacio de soluciones. Esta característica garantiza la localización del óptimo global, siempre que el espacio de búsqueda sea finito y computacionalmente tratable. Su principal limitación es la complejidad computacional, que crece exponencialmente con la dimensionalidad del problema, restringiendo su uso a problemas de escala reducida.

1.2 Conceptos fundamentales

- **Búsqueda exhaustiva:** Implica la evaluación de cada punto en el espacio de soluciones sin descartar ninguna región prematuramente. Esto asegura la identificación del óptimo global.
- **Espacio de búsqueda:** Es el conjunto de todas las soluciones candidatas. Puede ser de naturaleza discreta o continua. Para la implementación computacional, los espacios continuos deben ser discretizados.
- **Criterio de evaluación (función objetivo):** Es la función $f(\cdot)$ que se desea minimizar o maximizar. Permite cuantificar la calidad de cada solución candidata.
- **Complejidad computacional:** Para un problema con n variables, donde cada uno puede tomar k valores, la complejidad es de orden $O(k^n)$. Este crecimiento exponencial es conocido como la "maldición de la dimensionalidad".

1.3 Implementación para problemas con dominio discreto

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

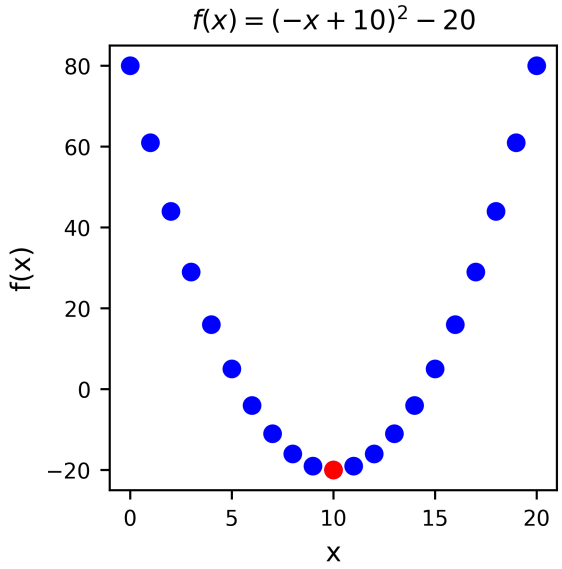
[2]:
```
resolucion = 600
```

[3]:
```
# ---------------------------------
# Encontrar el mínimo de la función:
```

```python
# f(x) = (-x + 10) ** 2 - 20
# x = {0, 1, 2, ..., 20}
# ----------------------------------

# Definición de la función objetivo
def f_discreta(x):
    return (-x + 10) ** 2 - 20

# Definir el espacio de búsqueda discreto
x = np.arange(0, 21, 1)

# Evaluar la función en todo el dominio (búsqueda exhaustiva)
y = f_discreta(x)

# Identificar el índice del valor mínimo
minimo_indice = np.argmin(y)

# Obtener la solución óptima
x_min = x[minimo_indice]
y_min = y[minimo_indice]

# Mostrar los resultados
print("x =", x_min)
print("f(x) =", y_min)
```

```
x = 10
f(x) = -20
```

[4]:
```python
# Graficar de la función
plt.figure(figsize = (3, 3), dpi = resolucion)
plt.scatter(x, y, color = "blue")

plt.title("$f(x) = (-x + 10)^{2} - 20$", fontsize = 10)
plt.xlabel("x", fontsize = 10)
plt.ylabel("f(x)", fontsize = 10)

# Añadir el punto óptimo
plt.scatter(x_min, y_min, color = "red")
plt.tick_params(labelsize = 8)
plt.show()
```



1.4 Adaptación para dominios continuos

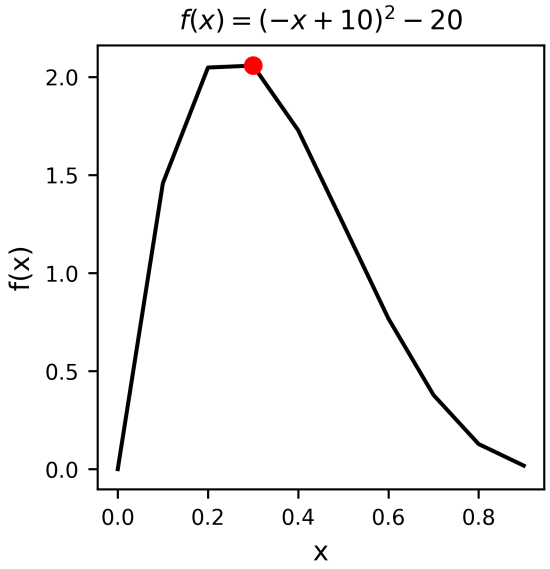
Para dominios continuos, es necesario discretizar el espacio de búsqueda. Este proceso introduce un error de aproximación controlado por la granularidad de la discretización (el tamaño del paso).

[5]:
```
# ---------------------------------
# Encontrar el máximo de la función:
# f(x) = 20 * x * (1 - x) ** 3
# x = (0, 1)
# ---------------------------------

# Definición de la función objetivo
def f_continua(x):
    return 20 * x * (1 - x) ** 3

# Discretizar el dominio continuo
step = 0.1
x = np.arange(0, 1, step)

# Evaluar la función en todo el dominio (búsqueda exhaustiva)
y = f_continua(x)

# Identificar el índice del valor máximo
maximo_indice = np.argmax(y)

# Obtener la solución óptima
x_max = x[maximo_indice]
y_max = y[maximo_indice]
```

```
# Mostrar los resultados
print("x =", x_max)
print("f(x) =", y_max)
```

```
x = 0.30000000000000004
f(x) = 2.058
```

[6]:
```
# Gráfico de la función
plt.figure(figsize = (3, 3), dpi = resolucion)
plt.plot(x, y, color = "black", zorder = 1)

plt.title("$f(x) = (-x + 10)^{2} - 20$", fontsize = 10)
plt.xlabel("x", fontsize = 10)
plt.ylabel("f(x)", fontsize = 10)

# Añadir el punto óptimo
plt.scatter(x_max, y_max, color = "red", zorder = 2)
plt.tick_params(labelsize = 8)
plt.show()
```



1.5 Extensión a problemas multivariados

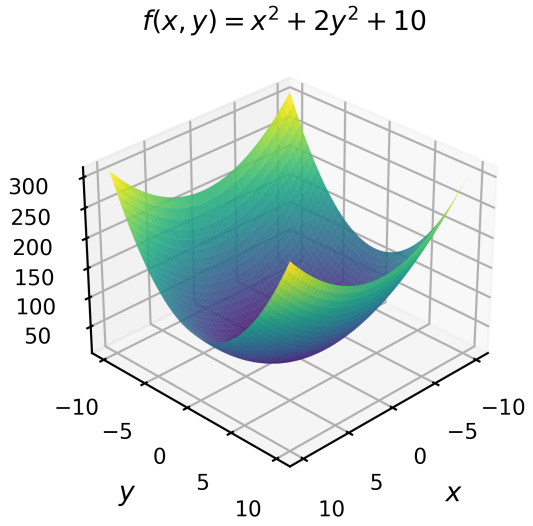
La generación a múltiples variables requiere la construcción de una grilla multidimensional que represente el espacio de búsqueda.

[7]:
```
# ---------------------------------
# Encontrar el mínimo de la función:
# f(x, y) = x ** 2 + 2 * y ** 2 + 10
# (x, y) = (-10, 10)
# ---------------------------------

# Definición de la función objetivo
def f_multi(x, y):
    return x ** 2 + 2 * y ** 2 + 10

# Discretizar el dominio continuo
step = 2
x = np.arange(-10, 10 + step, step)
y = np.arange(-10, 10 + step, step)

# Construir una grilla del espacio de búsqueda
X, Y = np.meshgrid(x, y)

# Evaluar la función objetivo
Z = f_multi(X, Y)

# Identificar el índice del valor mínimo
minimo_indice = np.argmin(Z)
minimo_2d = np.unravel_index(minimo_indice, Z.shape)

# Obtener la solución óptima
x_min = X[minimo_2d]
y_min = Y[minimo_2d]
z_min = Z[minimo_2d]

# Mostrar los resultados
print("x = ", x_min)
print("y = ", y_min)
print("f(x, y) = ", z_min)
```

```
x =  0
y =  0
f(x, y) =  10
```

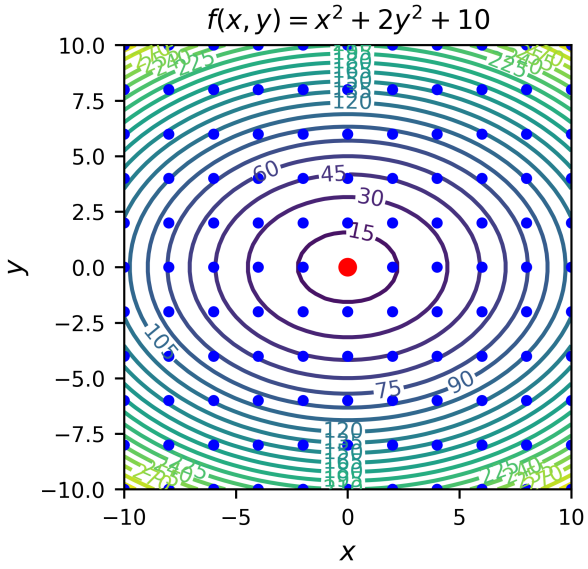
[8]:
```
# Graficar la función
x_valores = np.linspace(-10, 10, 50)
y_valores = np.linspace(-10, 10, 50)
X_valores, Y_valores = np.meshgrid(x_valores, y_valores)
Z_valores = f_multi(X_valores, Y_valores)

fig = plt.figure(figsize = (3, 3), dpi = resolucion)
ax = fig.add_subplot(111, projection = "3d")
```

```
ax.plot_surface(X_valores, Y_valores, Z_valores, cmap = "viridis", alpha = 0.9)
ax.view_init(elev = 30, azim = 45)

ax.set_title("$f(x, y) = x^2 + 2y^2 + 10$", fontsize = 10)
ax.set_xlabel("$x$", fontsize = 10)
ax.set_ylabel("$y$", fontsize = 10)
ax.set_zlabel("$f(x, y)$", fontsize = 10)
ax.tick_params(labelsize = 8)
plt.show()
```



[9]:
```
# Graficar la función
x_valores = np.linspace(-10, 10, 50)
y_valores = np.linspace(-10, 10, 50)
X_valores, Y_valores = np.meshgrid(x_valores, y_valores)
Z_valores = f_multi(X_valores, Y_valores)

plt.figure(figsize = (3, 3), dpi = resolucion)
contour = plt.contour(X_valores, Y_valores, Z_valores, levels = 20, cmap =␣
 ↪"viridis")

plt.clabel(contour, inline = True, fontsize = 8)
plt.title("$f(x, y) = x^2 + 2y^2 + 10$", fontsize = 10)
plt.xlabel("$x$", fontsize = 10)
plt.ylabel("$y$", fontsize = 10)

# Conjunto discreto de puntos
plt.scatter(X, Y, color = "blue", s = 10, zorder = 2)
plt.scatter(x_min, y_min, color = "red", zorder = 3)
```

```
plt.tick_params(labelsize = 8)
plt.show()
```



1.6 La maldición de la dimensionalidad

El principal obstáculo de la fuerza bruta es el crecimiento exponencial del espacio de búsqueda con el número de variables. Consideremos la minimización de una función de 10 variables.

$$f(x_1, x_2, \ldots, x_{10}) = \sum_{i=1}^{10} (x_i - i)^2,\ x_i \in [-10, 10]$$

El objetivo es encontrar el conjunto de valores $(x_1, x_2, \ldots, x_{10})$ que minimizan la función f. Si discretizamos cada variable con una paso de 0.1 en el intervalo $[-10, 10]$, obtenemos:

$$\text{Puntos por variable} = \frac{(10 - (-10))}{0.1} + 1 = 201$$

Luego, el número total de combinaciones es:

$$\text{Total de combinaciones} = 201^{10} \approx 2.57 \times 10^{23}$$

Ahora supongamos, de forma optimista, que nuestro computador puede evaluar un millón (10^6) de combinaciones por segundo. Entonces, el tiempo total para evaluar todas las combinaciones es:

$$\frac{2.57 \times 10^{23} \text{ combinaciones}}{10^6 \text{combinaciones / seg}} = 2.57 \times 10^{17} \text{ segundos}$$

Si este resultado lo convertimos en años:

$$\frac{2.57 \times 10^{17} \text{ segundos}}{3.154 \times 10^{7} \text{segundos / año}} \approx 8.15 \times 10^{9} \text{ años}$$

Por lo tanto, tomaría aproximadamente 8150 millones de años completar la evaluación.

Esta limitación fundamental hace que el algoritmo de fuerza bruta sea viable únicamente para problemas de baja dimensionalidad o con espacios de búsqueda muy restringidos. Para problemas de mayor escala, es necesario recurrir a algoritmos más sofisticados que exploten la estructura del problema o utilicen estrategias heurísticas para reducir el espacio de búsqueda efectivo.

2 Busquedas aleatorias

2.1 Principios fundamentales

El algoritmo de búsqueda aleatorias representa una alternativa estocástica al enfoque determinista de la fuerza bruta. En lugar de evaluar sistemáticamente todas las soluciones posibles, este método genera un conjunto de soluciones candidatas mediante procesos aleatorios, evaluándolas para identificar la mejor encontrada durante el proceso de búsqueda.

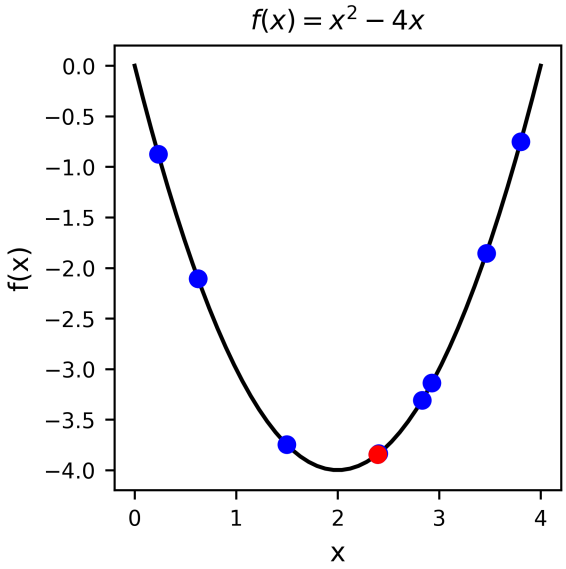
Esta estrategia sacrifica la garantía de encontrar el óptimo global a cambio de una reducción significativa en el costo computacional. La efectividad del método depende tanto del número de muestras generadas como de la distribución de probabilidad utilizada para el muestreo del espacio de búsqueda.

[10]:
```python
# ---------------------------------
# Encontrar el mínimo de la función:
# f(x) = x ** 2 - 4 * x
# x = [0, 4]
# ---------------------------------

# Definición de la función objetivo
def f_uni_aleatorio(x):
    return x ** 2 - 4 * x

# Definir el espacio de búsqueda aleatorio
np.random.seed(42)
n = 10
x = np.random.uniform(0, 4, n)

# Evaluar la función en todo el dominio
y = f_uni_aleatorio(x)

# Identificar el índice del valor mínimo
minimo_indice = np.argmin(y)

# Obtener la solución óptima
x_min = x[minimo_indice]
```

```
y_min = y[minimo_indice]

# Mostrar los resultados
print("x =", x_min)
print("f(x) =", y_min)
```

```
x = 2.3946339367881464
f(x) = -3.8442640559350894
```

[11]:
```
# Gráfico de la función
x_valores = np.linspace(0, 4, 50)
y_valores = f_uni_aleatorio(x_valores)

plt.figure(figsize = (3, 3), dpi = resolucion)
plt.plot(x_valores, y_valores, color = "black", zorder = 1)

plt.title("$f(x) = x^2 - 4x$", fontsize = 10)
plt.xlabel("x", fontsize = 10)
plt.ylabel("f(x)", fontsize = 10)

# Añadir el punto óptimo
plt.scatter(x, y, color = "blue", zorder = 2)
plt.scatter(x_min, y_min, color = "red", zorder = 3)
plt.tick_params(labelsize = 8)
plt.show()
```



2.2 Consideraciones sobre la convergencia

La convergencia del algoritmo de búsquedas aleatorias hacia el óptimo global es probabilística y depende del número de muestras generadas. Según el teorema del límite central, cuando el númro de muestras tiende a infinito, la probabilidad de encontrar una solución arbitratiamente cercana al óptimo global tiende a uno, siempre que la distribución de muestreo tenga soporte en todo el espacio de búsqueda.

En la práctica, el número de muestras requerido para alcanzar una aproximación satisfactoria del óptimo puede ser considerablemente menor que el número total de puntos en una discretización fina del espacio, especialmente en problemas de alta dimensionalidad donde la búsqueda exhaustiva es computacionalmente prohibitiva.

Juan F. Olivares Pacheco (`jfolivar@uda.cl`)

Universidad de Atacama, Facultad de Ingeniería, Departamento de Matemática