Manual – Búsqueda y Optimización

Algoritmos de Búsqueda

Un algoritmo de búsqueda es una técnica que permite explorar un espacio de estados, normalmente representado como un grafo, con el fin de llegar desde un estado inicial hasta un estado meta. Para que un problema pueda ser modelado como búsqueda, se tienen que tener en cuenta:

- **Estado inicial:** punto de partida.
- **Estado meta:** objetivo.
- **Reglas de producción:** cómo se generan nuevos estados (sucesores).
- **Reglas de control:** cómo se eligen los estados a expandir.

Los algoritmos de búsqueda se pueden clasificar en dos grandes grupos:

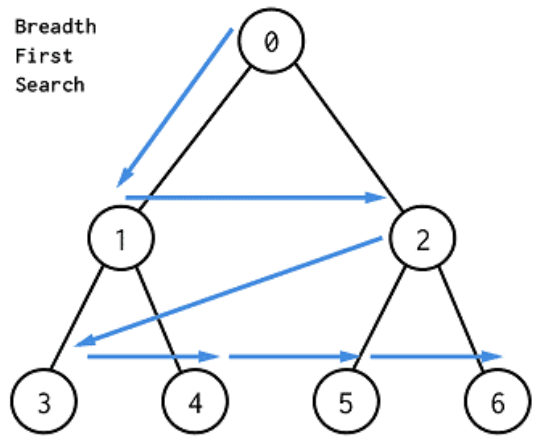
- Búsqueda no informada (ciega): no usa información adicional, solo explora sistemáticamente hasta hallar la meta.
- Búsqueda informada (con heurística): utiliza conocimiento extra para guiar la exploración de forma más eficiente.

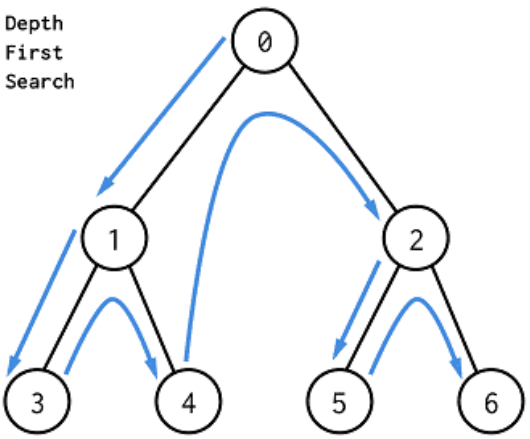
Búsqueda No Informada

- **BFS:** explora nivel por nivel (cola FIFO).
- **DFS:** explora en profundidad (pila LIFO).
- **UCS:** elige siempre el camino de menor costo acumulado (cola de prioridad).

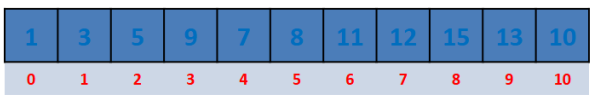
Comparación rápida

Algoritmo	Estrategia	Garantiza	Pros	Contras
BFS	Nivel por nivel (cola FIFO)	Ruta más corta en pasos	Siempre encuentra la solución mínima en pasos	Mucha memoria
DFS	Profundidad (pila LIFO)	No	Poca memoria	Puede quedarse en ramas largas
UCS	Menor costo acumulado (heap)	Ruta de menor costo	Óptimo en costo	Lento y pesado

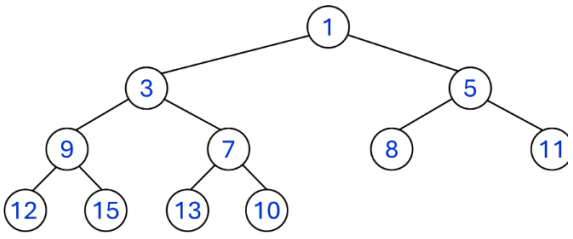




Así se almacena en tu computador....



.....Pero así te lo debes imaginar!



BFS

```
def BFS(nodoInicial):
    ABIERTOS = [nodoInicial]   # frontera
```

```
    CERRADOS = []            # visitados
    exito = False
    fracaso = False

    while not exito and not fracaso:
        nodoActual = extraePrimero(ABIERTOS)  # saca el más antiguo (FIFO)
        CERRADOS = ingresaFinal(CERRADOS, nodoActual)

        if esMeta(nodoActual):
            exito = True
        else:
            listaSucesores = sucesores(nodoActual, ABIERTOS, CERRADOS)
            for nodo in listaSucesores:
                ABIERTOS = ingresaFinal(ABIERTOS, nodo)  # se agrega al final

        if ABIERTOS == []:
            fracaso = True

    if exito:
        return Solucion(nodoActual, nodoInicial)
    else:
        return None
```

- Recorre nivel por nivel.
- Útil cuando quieres la solución más corta en cantidad de pasos, sin importar el costo.
- Ejemplo: encontrar la salida más cercana en un laberinto donde todos los caminos cuestan lo mismo.
- Problema: gasta mucha memoria, porque guarda todos los nodos de cada nivel.

DFS

```
def DFS(nodoInicial):
    ABIERTOS = [nodoInicial]
    CERRADOS = []
```

```
    exito = False
    fracaso = False

    while not exito and not fracaso:
        nodoActual = extraePrimero(ABIERTOS)  # saca el más reciente
        CERRADOS = ingresaFinal(CERRADOS, nodoActual)

        if esMeta(nodoActual):
            exito = True
        else:
            listaSucesores = sucesores(nodoActual, ABIERTOS, CERRADOS)
            for nodo in listaSucesores:
                ABIERTOS = ingresaInicio(ABIERTOS, nodo)  # se mete al inicio (p
ila)

        if ABIERTOS == []:
            fracaso = True

    if exito:
        return Solucion(nodoActual, nodoInicial)
    else:
        return None
```

- Se mete “hacia abajo” en un camino hasta que ya no puede, y recién ahí retrocede.
- Consume poca memoria, porque solo guarda el camino actual.
- Bueno si se quiere llegar rápido a una solución cualquiera, sin importar si es óptima.
- Ejemplo: resolver un rompecabezas pequeño probando opciones en cadena.
- Problema: puede perderse en caminos largos y nunca encontrar la mejor solución.

UCS

```
def UCS(nodoInicial):
    ABIERTOS = [nodoInicial]
    CERRADOS = []
    exito = False
    fracaso = False

    while not exito and not fracaso:
        nodoActual = desencolar(ABIERTOS)  # saca el de menor costo (heap
q.heappop)
        CERRADOS = ingresaFinal(CERRADOS, nodoActual)

        if esMeta(nodoActual):
            exito = True
        else:
            listaSucesores = sucesores(nodoActual, ABIERTOS, CERRADOS)
            for nodo in listaSucesores:
                ABIERTOS = encolar(ABIERTOS, nodo)  # heapq.heappush

        if ABIERTOS == []:
            fracaso = True

    if exito:
        return Solucion(nodoActual, nodoInicial)
    else:
        return None
```

- Expande siempre el nodo con menor costo acumulado.
- Garantiza la ruta más barata, aunque no sea la más corta en pasos.
- Ejemplo: encontrar el camino más barato en un mapa donde cada calle tiene un costo distinto.
- Problema: puede ser más lento, porque revisa muchas alternativas antes de llegar a la meta.

Reconstrucción de la solución

```
def Solucion(nodo, inicial):
    solucion = []
    while nodo is not inicial:
        solucion = [str(nodo)] + solucion
        nodo = nodo.padre
    return [str(inicial)] + solucion
```

- Permite armar el camino desde el inicio hasta la meta siguiendo los padres de cada nodo.
- Básicamente, “retrocede hacia atrás” hasta el estado inicial y devuelve la secuencia de pasos.
- Esto es común en todos los algoritmos de búsqueda, porque encontrar la meta no basta: se necesita también mostrar cómo se llegó a esa meta.

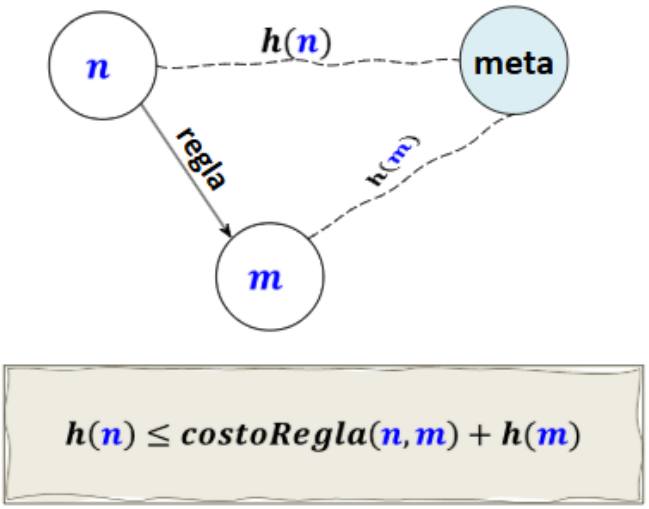
Búsqueda Informada y Heurísticas

Cuando el espacio de búsqueda es muy grande, conviene usar heurísticas, es decir, reglas o funciones que estiman qué tan cerca estamos de la meta.

- Una heurística admisible nunca sobreestima el costo real (es optimista).
- Una heurística consistente cumple la desigualdad triangular (”es logica”):

$$h(n) \leq costo(n, m) + h(m)$$

y siempre vale 0 en la meta.



Algoritmo A*

El algoritmo **A*** es una técnica de búsqueda **informada** que combina lo mejor de la búsqueda de costo uniforme (UCS) y de la búsqueda heurística. La idea es expandir siempre el nodo que parece más prometedor, usando la función:

$$f(n) = g(n) + h(n)$$

- **g(n):** costo real desde el estado inicial hasta `n` (cuántos movimientos se han hecho).
- **h(n):** estimación heurística del costo que falta desde `n` hasta la meta.
- **f(n):** costo total estimado de pasar por `n` hasta llegar a la meta.

Clase nodo

Cada estado del puzzle se representa como un **nodo**, que guarda:

- `puzzle`: matriz con el estado actual.
- `pos0`: posición del espacio vacío.
- `padre`: para reconstruir la solución.

- `costo`: número de movimientos desde el inicio (`g(n)`).
- `f`: valor de evaluación (`g + h`).

```
class nodo:
    def __init__(self, puzzle, pos0, padre, costo):
        self.puzzle = copy(puzzle)
        self.pos0 = pos0
        self.padre = padre
        self.costo = costo
        self.f = self.costo + self.heuristica2()  # f(n) = g + h
```

Heurísticas implementadas

1. `heuristica1`: número de fichas fuera de lugar.
2. `heuristica2`: distancia de Manhattan (más precisa y común).

```
def heuristica2(self):
    meta = array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
    distancia = 0
    for numero in range(1, 9):
        f, c = where(self.puzzle == numero)
        fmeta, cmeta = where(meta == numero)
        distancia += abs(f[0] - fmeta[0]) + abs(c[0] - cmeta[0])
    return distancia
```

Función de solución

Para reconstruir el camino desde el inicio hasta la meta, se siguen los punteros `padre`.

```
def Solucion(nodo, inicial):
    solucion = []
    while nodo is not inicial:
        solucion = [str(nodo)] + solucion
```

```
        nodo = nodo.padre
    return [str(inicial)] + solucion
```

A*

La búsqueda se maneja con dos listas:

- ABIERTOS (frontera): estados por explorar, ordenados como heap según `f(n)`.
- CERRADOS: estados ya explorados.

```
def Aestrella(nodoInicial):
    ABIERTOS = []
    heappush(ABIERTOS, nodoInicial)
    CERRADOS = []
    exito = False
    fracaso = False
    cont = 0

    while not exito and not fracaso and cont <= MAX:
        nodoActual = heappop(ABIERTOS)   # saca el de menor f
        CERRADOS.append(nodoActual)

        if nodoActual.esMeta():
            exito = True
        else:
            listaSucesores = nodoActual.sucesores(ABIERTOS, CERRADOS)
            for nodo in listaSucesores:
                heappush(ABIERTOS, nodo)

            if ABIERTOS == []:
                fracaso = True
        cont += 1

    if exito:
        return Solucion(nodoActual, nodoInicial), cont
    else:
```

```
        return None, cont
```

1. Extrae el nodo con menor `f`.
2. Si es meta → termina.
3. Si no → genera sucesores y los mete en `ABIERTOS`.
4. Repite hasta encontrar solución o llegar al límite de iteraciones.

Cuándo usar

- Problemas de **búsqueda de rutas** (laberintos, GPS, rompecabezas).
- Cuando hay una buena heurística disponible.
- Se usa cuando necesitamos **garantizar el óptimo** en costo o en pasos.

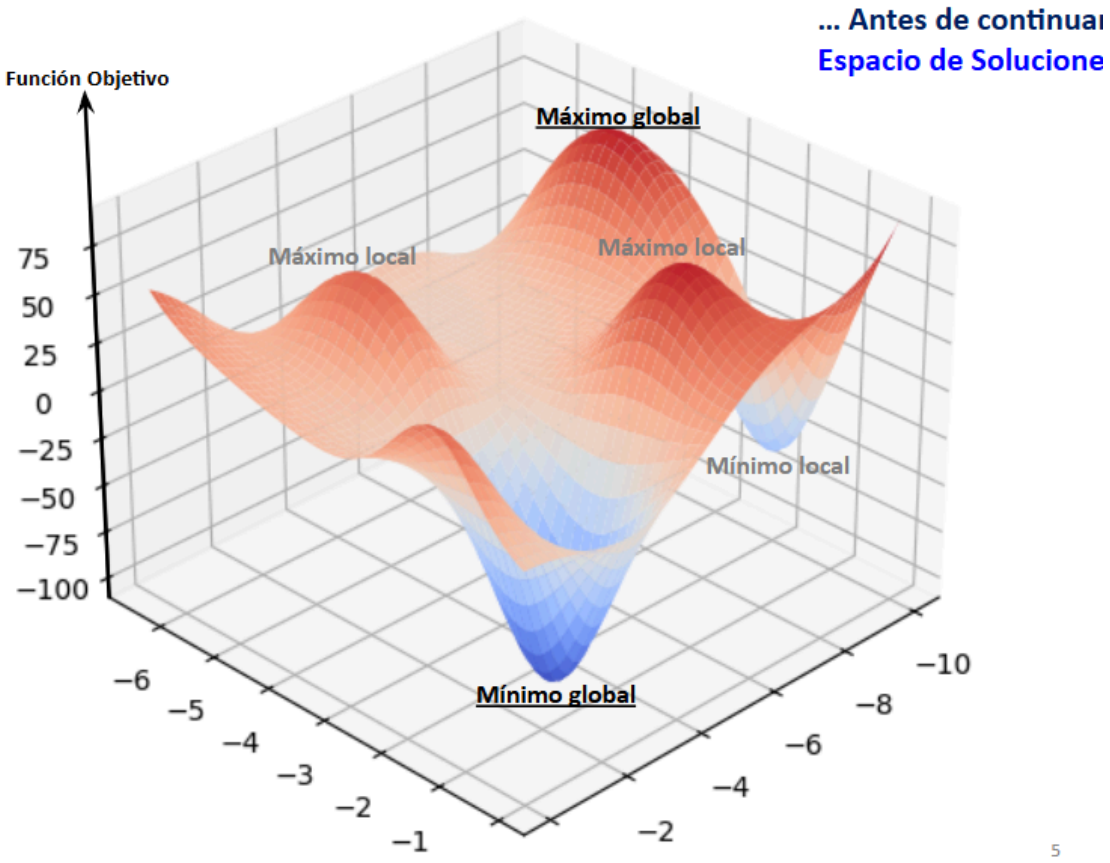
Metaheurísticas

Las metaheurísticas son algoritmos de optimización aproximados que se usan cuando el espacio de búsqueda es demasiado grande o complejo para explorarlo exhaustivamente.

- No recorren todo el espacio de soluciones.
- Equilibran exploración (buscar en nuevas zonas) y explotación (refinar buenas soluciones).
- Encuentran soluciones muy buenas en tiempo razonable, aunque no siempre el óptimo global.

Se tiene que tener en cuenta:

- **Función objetivo:** mide la calidad de una solución.
- **Vecindad:** soluciones cercanas entre sí.



Algoritmos Genéticos (AG)

Los Algoritmos Genéticos son metaheurísticas inspiradas en la evolución biológica. La idea es que cada solución del problema se representa como un individuo y que, a través de operadores que imitan la naturaleza (selección, cruzamiento, mutación, elitismo), la población de soluciones va evolucionando hacia cada vez mejores resultados.

Representación de un Individuo

Un individuo es una posible solución al problema.

- En el caso de la Mochila, se usa un vector binario:
  - Cada posición corresponde a un objeto.
  - `1` significa que el objeto está en la mochila.
  - `0` significa que el objeto no está.
- Al final del vector se guarda el fitness (beneficio total de esa solución).

```
Individuo = [1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 290]
```

```
def generaIndividuo(n, B, P, C):
    individuo = [randint(0,1) for _ in range(n)]
    individuo.append(fitness(individuo, n, B, P, C))
    return individuo
```

Función de Fitness

El fitness mide qué tan buena es una solución.

- Se calcula sumando los beneficios (`B`) de los objetos elegidos.
- Si el peso (`P`) supera la capacidad (`C`), el fitness se hace `0` .

```
def fitness(individuo, n, B, P, C):
    sumValor, sumPeso = 0, 0
    for i in range(n):
        if individuo[i] == 1:
            sumValor += B[i]
            sumPeso += P[i]
    if sumPeso > C:
        return 0
    return sumValor
```

Operadores Genéticos

a) Selección

Proceso de elegir qué individuos serán padres para la siguiente generación.

La idea es que los mejores tengan más chances.

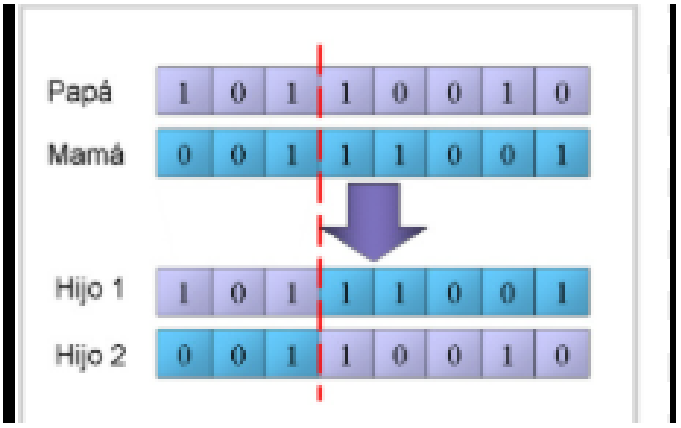
```
def seleccion(poblacion, n):
    posibles = sample(poblacion, k)
    posibles.sort(key=lambda x: x[n], reverse=True)
    return [posibles[0], posibles[1]]  # los 2 mejores
```

b) Cruzamiento

Combina dos padres para generar hijos.

- Se elige un punto de cruce.
- Se toma la primera parte del padre1 y la segunda del padre2.
- Luego se recalcula el fitness del hijo.

```
def cruzamiento(indiv1, indiv2, n, B, P, C):
    ptoCruce = randint(1, n-1)
    hijo = indiv1[:ptoCruce] + indiv2[ptoCruce:]
    hijo[n] = fitness(hijo, n, B, P, C)
    return hijo
```



c) Mutación

Introduce variación aleatoria cambiando genes de un individuo.

- Recorre todos los genes.
- Con una probabilidad (`tasaMutacion`) cambia un `0` a `1` o viceversa.

```
def mutacion(individuo, n, B, P, C):
    for i in range(n):
        if random() < tasaMutacion:
            individuo[i] = 1 - individuo[i]
    individuo[n] = fitness(individuo, n, B, P, C)
```

d) Elitismo

Asegura que los mejores individuos de una generación pasen directamente a la siguiente sin ser modificados.

```
if conElitismo:
    nuevaPoblacion = poblacion[0:numElegidos]
```

Ciclo del Algoritmo Genético

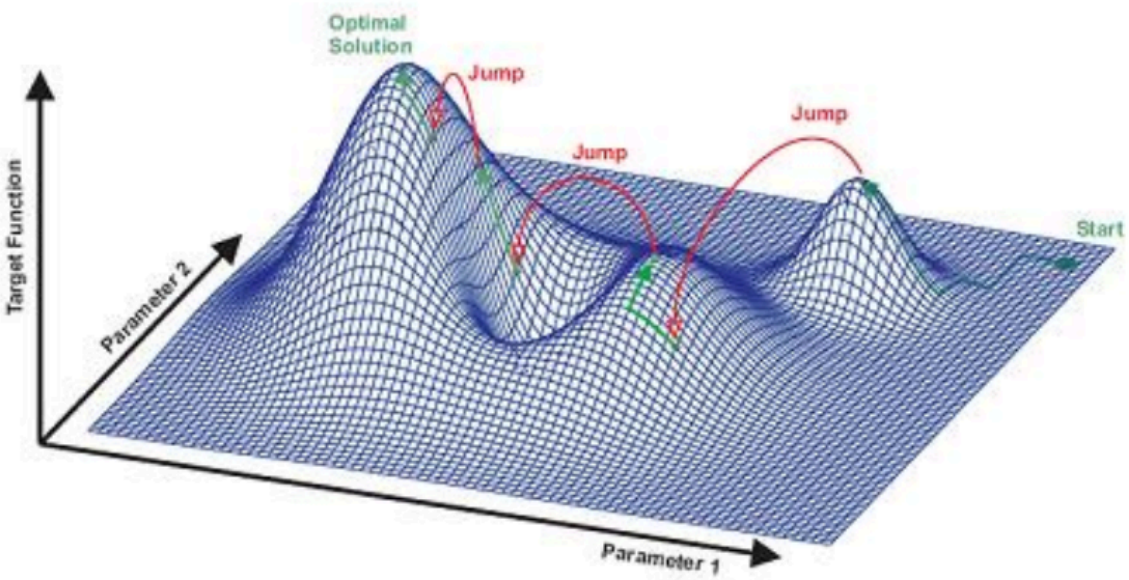
1. Generar población inicial (soluciones al azar).
2. Evaluar fitness de cada individuo.
3. Seleccionar padres (mejores soluciones).
4. Cruzamiento y mutación → nuevos individuos.
5. Elitismo → mantener los mejores.
6. Repetir el ciclo hasta cumplir el número de generaciones.
7. Retornar el mejor individuo encontrado.

Simulated Annealing (SA)

El Recocido Simulado es una metaheurística inspirada en el proceso de enfriamiento del metal. Explora soluciones vecinas aceptando no solo mejoras,

sino también peores soluciones con cierta probabilidad, lo que permite escapar de óptimos locales.

... Antes de continuar: Espacio de Soluciones



Representación de una Solución

Una solución es un vector binario que indica qué objetos están en la mochila:

- `1` → objeto incluido.
- `0` → objeto no incluido.

```
Solución = [1, 0, 1, 1, 0, 0, 1, 0, 1, 0]
```

Función Objetivo

Mide el beneficio total de la solución.

Si el peso supera la capacidad `C`, el valor es `0`.

```
def funcionObjetivo(solucion, B, P, C, n):
    valorTotal, pesoTotal = 0, 0
```

```
    for i in range(n):
        if solucion[i] == 1:
            valorTotal += B[i]
            pesoTotal += P[i]
    if pesoTotal > C:
        return 0
    return valorTotal
```

Función para Generar un Vecino

Se genera cambiando un bit al azar (agregar o quitar un objeto).

```
def solucionVecina(solucion, n):
    nuevaSolucion = solucion[:]
    pos = randint(0, n-1)
    nuevaSolucion[pos] = 1 - nuevaSolucion[pos]
    return nuevaSolucion
```

Probabilidad de Aceptación

- Si el vecino es mejor, se acepta.
- Si es peor, se acepta con probabilidad:

$$P = e^{\frac{valorNuevo - valorActual}{T}}$$

```
def probabilidadAceptacion(valorActual, valorNuevo, T):
    if valorNuevo > valorActual:
        return 1.0
    else:
        return exp((valorNuevo - valorActual)/T)
```

Ciclo del Algoritmo

1. Generar solución inicial.

2. Evaluar su beneficio.
3. Mientras la temperatura > mínima:
    - Generar vecino.
    - Decidir si se acepta.
    - Actualizar mejor solución.
    - Reducir temperatura (`T = T * tasaEnfriamiento`).
4. Retornar la mejor solución encontrada.

Tabú Search (TS)

La Búsqueda Tabú explora todo el vecindario de una solución y elige el mejor vecino, evitando ciclos gracias a una lista tabú que prohíbe regresar a soluciones recientes.

Incluye un criterio de aspiración para permitir soluciones tabú si son mejores que la mejor global.

Representación de una Solución

Un vector binario con el beneficio al final:

```
Solución = [1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 250]
```

Función Objetivo

Misma que en AG y SA: suma valores de los objetos, invalida si excede `C`.

```
def funcionObjetivo(solucion, n, B, P, C):
    valorTotal, pesoTotal = 0, 0
    for i in range(n):
        if solucion[i] == 1:
            valorTotal += B[i]
            pesoTotal += P[i]
```

```
    if pesoTotal > C:
        return 0
    return valorTotal
```

Función para Generar la Vecindad

Todos los vecinos posibles se generan cambiando un bit del vector:

```
def generaVecindario(solucionActual, n, B, P, C):
    vecindario = []
    for i in range(n):
        vecino = solucionActual[:n]
        vecino[i] = 1 - vecino[i]
        vecino.append(funcionObjetivo(vecino, n, B, P, C))
        vecindario.append(vecino)
    return vecindario
```

```
Solución actual: [1, 0, 1, 0, 150]

Vecindario:
[0, 0, 1, 0, 90]
[1, 1, 1, 0, 200]
[1, 0, 0, 0, 70]
[1, 0, 1, 1, 170]
```

Ciclo del Algoritmo

1. Generar solución inicial.
2. Evaluar su beneficio.
3. Repetir hasta el máximo de iteraciones:
    - Generar vecindario completo.
    - Ordenar vecinos por beneficio.

- Elegir mejor vecino **no tabú** (o tabú si cumple aspiración).
- Actualizar lista tabú con la solución usada y su tiempo de tenencia.
- Actualizar mejor solución global.

4. Retornar la mejor solución encontrada.

Búsqueda Adversarial

Se aplica en juegos de dos jugadores con intereses opuestos (ajedrez, damas, gato), donde:

- Juegos de suma cero: lo que gana uno, lo pierde el otro.
- Deterministas: sin azar ni incertidumbre.
- Información perfecta: ambos ven todo el estado.

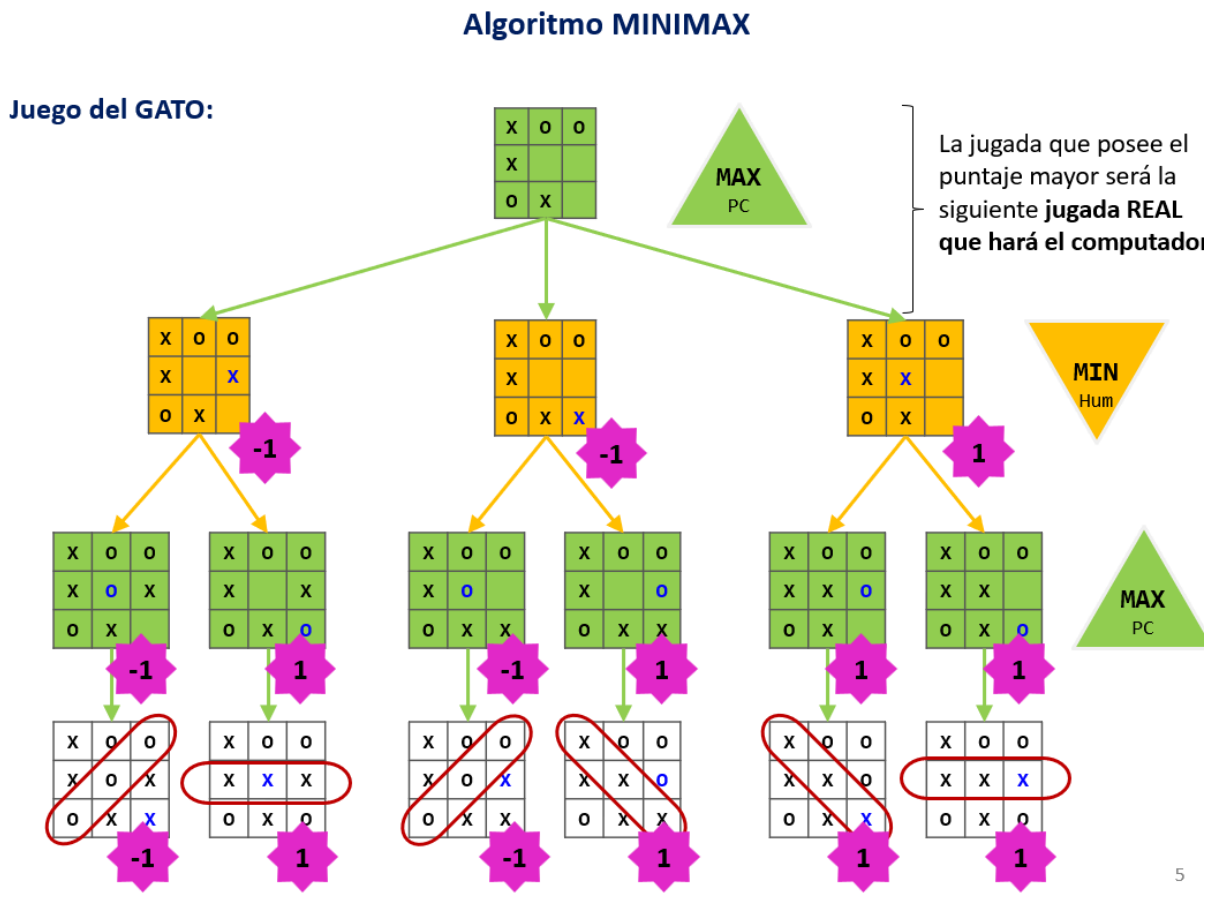
A diferencia de las metaheurísticas, aquí no se busca una buena solución aproximada, sino la mejor jugada posible asumiendo que el rival también jugará de forma óptima.

Algoritmo Minimax

- **MAX (computador):** intenta maximizar.
- **MIN (oponente):** intenta minimizar.
- Construye un árbol de juego donde se simulan todas las jugadas posibles.
- Los nodos hoja tienen valores:
  - +1 si gana MAX.
  - 1 si gana MIN.
  - 0 si es empate.
- La utilidad se propaga hacia arriba: MAX elige el máximo, MIN el mínimo.

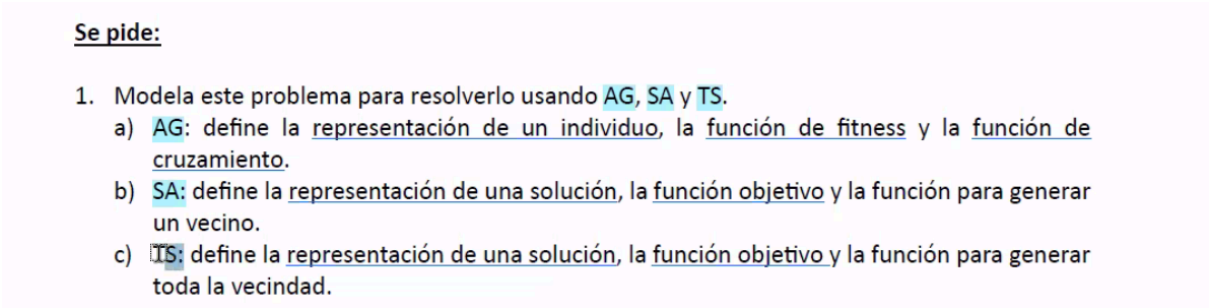
Ventaja: garantiza la mejor jugada.

Desventaja: el árbol crece exponencialmente (muy costoso en juegos complejos).



Técnica	Cuándo usarla	Ventajas	Limitaciones
BFS, DFS, UCS	Problemas sin optimización compleja	Simples, UCS da solución óptima en costo	Ineficientes en espacios grandes
A*	Problemas de rutas con buena heurística	Eficiente y óptimo	Depende de heurística, usa mucha memoria
AG	Optimización combinatoria	Evolución de soluciones, flexible	Parámetros sensibles
SA	Optimización compleja con óptimos locales	Escapa de óptimos locales	Ajuste de T y enfriamiento crítico

Técnica	Cuándo usarla	Ventajas	Limitaciones
TS	Problemas combinatorios grandes	Memoria evita ciclos, diversifica	Depende del vecindario/tabú
Minimax	Juegos adversariales (ajedrez, gato)	Garantiza mejor jugada	Árbol crece exponencialmente



Se pide:

1. Modela este problema para resolverlo usando AG, SA y TS.
   a) AG: define la representación de un individuo, la función de fitness y la función de cruzamiento.
   b) SA: define la representación de una solución, la función objetivo y la función para generar un vecino.
   c) TS: define la representación de una solución, la función objetivo y la función para generar toda la vecindad.

modelar un problema según la técnica que corresponda. Eso significa identificar bien los elementos del modelo y cómo se aplican en cada caso.

- Búsqueda no informada (no optimización, ej. BFS y DFS): hay que definir el estado inicial, el estado meta (si lo hay), las reglas de producción (cómo se generan los sucesores) y las reglas de control (cómo elegir qué expandir: FIFO para BFS, LIFO para DFS).
- Búsqueda no informada para optimización (ej. UCS): se definen también estado inicial, reglas de producción y reglas de control, pero en este caso la frontera se maneja con una cola de prioridad para ir expandiendo siempre el nodo de menor costo acumulado.
- Metaheurísticas (AG, SA, TS):
  - En AG lo importante es mostrar cómo representas los individuos y, en particular, cómo se realiza el cruzamiento entre soluciones.

- En SA hay que mostrar cómo generarías una solución vecina y cómo controlas la aceptación con la temperatura.
- En TS lo clave es explicar cómo generarías todas las vecindades de entorno y cómo funciona la lista tabú para evitar volver atrás.

Plantear un problema con sus estados y reglas, y luego aplicar la técnica adecuada (ya sea búsqueda o metaheurística), sabiendo qué parte del algoritmo es fundamental en cada caso.