Aproximaciones a problemas NP duros: Algoritmos heurísticos y aproximados

Algoritmos aproximados

Decimos que H es un algoritmo ε- aproximado para el problema Π si para algún ε > 0

$$x^H(I) - x^*(I) \mid \leq \varepsilon \mid x^*(I) \mid$$ (problema de minimización)

O equivalentemente un algoritmo es aproximado si existe $\rho > 0$ tal que para toda instancia I

$$x^H(I) / x^*(I) \leq \rho$$

Para x*= valor óptimo

Situación ideal, pero poco frecuente

Problemas NP-completos,

- Algoritmos heurísticos: Procedimientos que pueden producir una buena solución e incluso la solución óptima.

- Un método heurístico es un procedimiento para resolver un problema de optimización bien definido mediante una aproximación intuitiva, en la que la estructura del problema se utiliza de forma inteligente para obtener una buena solución.

- El tiempo invertido por un método exacto para encontrar la solución óptima es inmenso, si es que existe tal método.

  Es de un orden de magnitud muy superior al del heurístico (pudiendo llegar a ser tan grande en muchos casos, que sea inaplicable).

- Las heurísticas atacan la complejidad guiando la búsqueda por los caminos “más prometedores “ en el espacio de búsqueda.

- Encuentran soluciones aceptables.

- Desafortunadamente las reglas son falibles, pues usan información limitada.

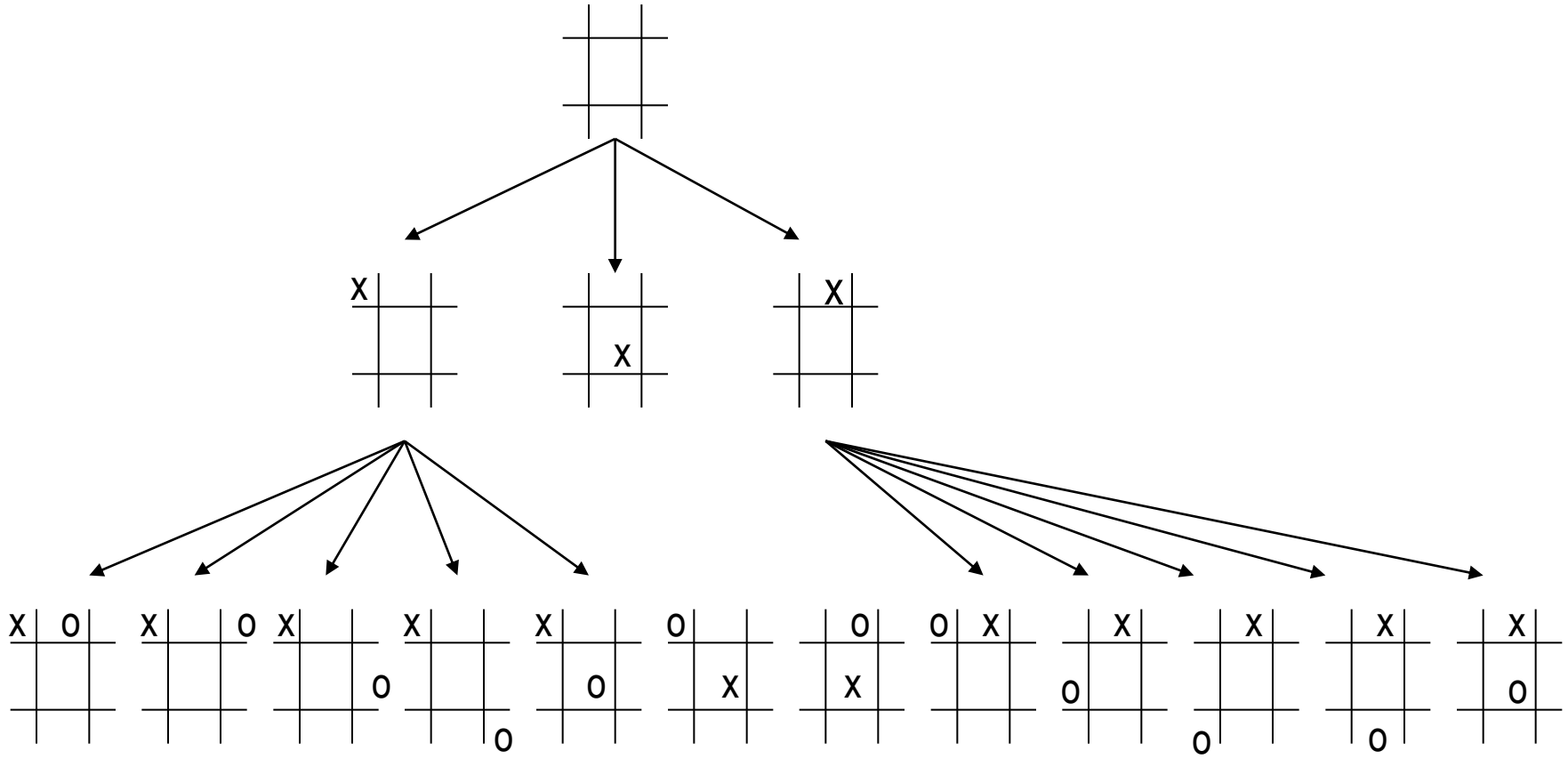
- No es posible examinar cada inferencia que puede ser hecha en un dominio matemático, o cada movimiento que puede ser hecho en un tablero de ajedrez.

- Heurísticas componente esencial para esos casos

Se puede pensar que los algoritmos heurísticos constan de dos partes:

1. La medida heurística.

2. Un algoritmo que la usa para buscar en el espacio de soluciones.



Algoritmos Heurísticos

- Usan **funciones de evaluación** y se detienen al conseguir una *buena* solución
- Algoritmos heurísticas
  - Minimizar costo estimado para alcanzar objetivo desde donde estoy
  - Ejemplo: Algoritmo de Búsqueda "*Primero el Mejor para expandir*"
- **Clasificación**
  - Iterativo (Descenso de Gradiente, Algoritmos Genéticos, Temple Simulado) vs. Constructivo (Primero el Mejor)
  - Búsqueda Local (Temple Simulado) vs. Búsqueda Global (Algoritmos Genéticos)

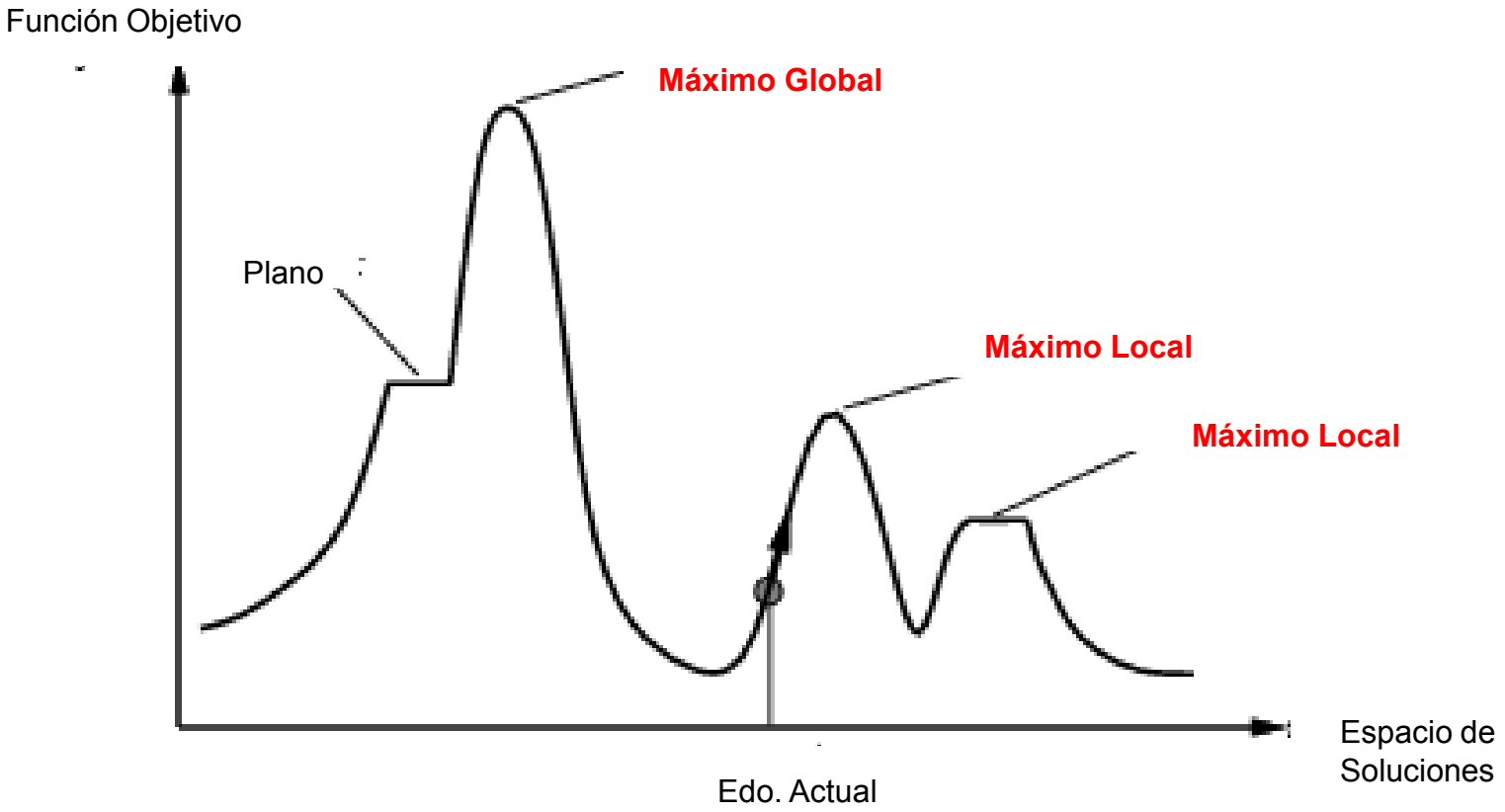
Algoritmos Heurísticos

- Algoritmo Heurística General (**CONTRUCTIVO**)
  - Crear árbol de búsqueda solo con nodo raíz
  - Crear una lista de vecinos al nodo actual para expandir
  - Seleccionar cada nodo de la lista y evaluar solución parcial (uso **Función de Evaluación**)
  - Escoger para **expandir mejor sucesor** y repetir si no es edo. objetivo

- Hillclimb (**ITERATIVO**: mejora solución inicial)
  - Actual = Solución-Inicial(problema)
  - Lazo hasta converger
    - próximo= solución vecina a la actual (problema)
    - Si valor(actual) >valor(próximo)
      - actual=próximo

Búsqueda Local

- Problema: cuidado con los máximos locales



Primero el Mejor (constructivo)

- Idea: usar una función de evaluación *f(n) p*or nodo
  - Estima que tan "deseable“ es

- Algoritmo:
- Tomar nodo actual
  - **Evaluar** los nodos sucesores
  - Si alguno es el estado final
    - Detenerse
  - de lo contrario
    - Expandir nodo no expandido **mas deseable**
  - Regresar al paso inicial hasta no tener a quien expandir

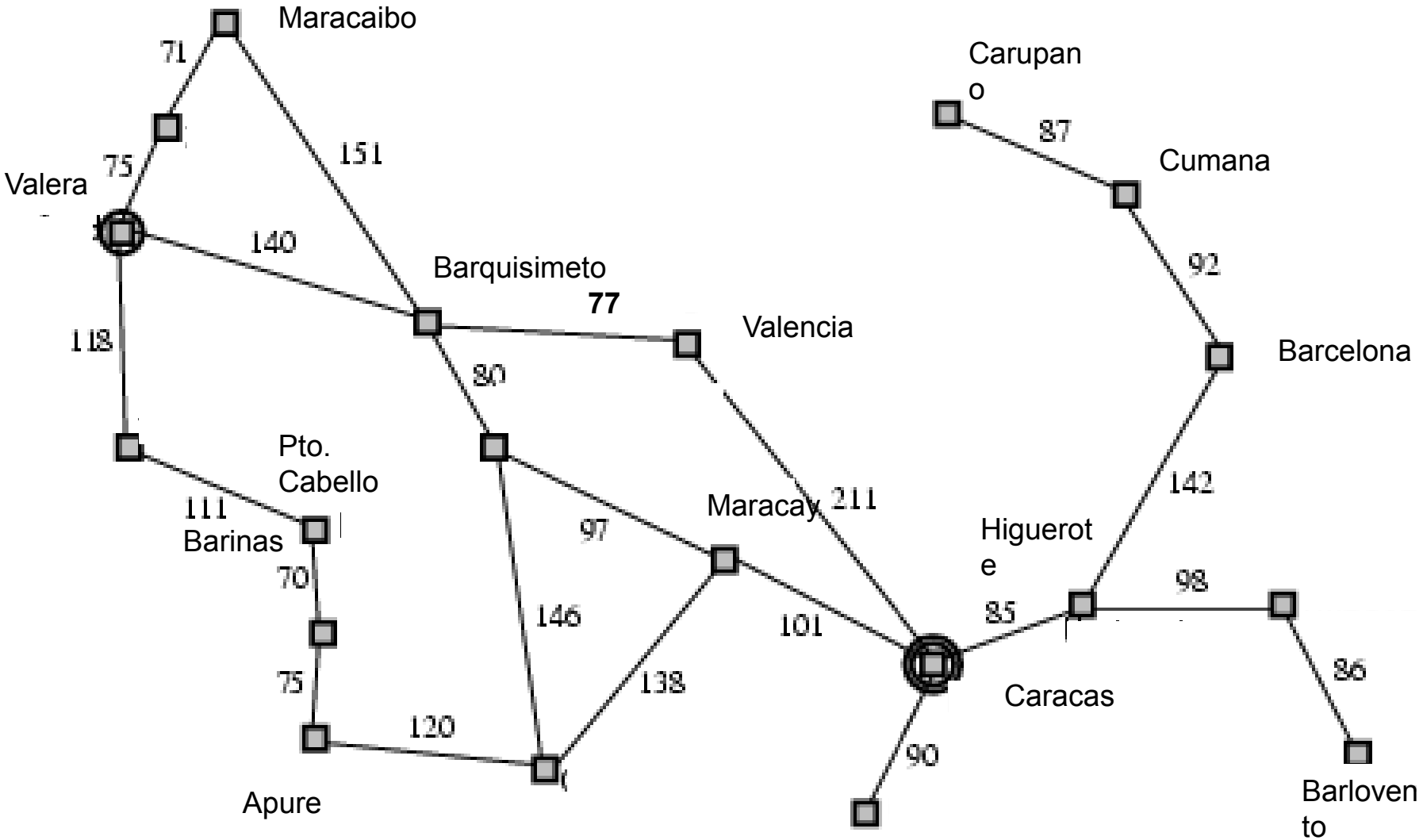
Primero el Mejor

Ejemplo: problema del viajero

- **Función de evaluación** *f(n)*= estima costo desde *n* hasta *objetivo*

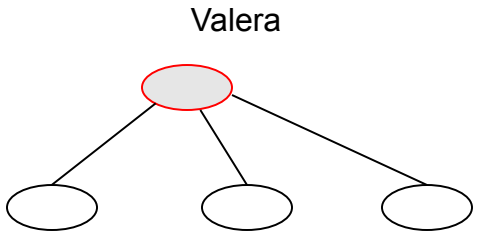
  Por ejemplo: *f(n)* = distancia desde *n* a Caracas
- Algoritmo **expande nodo** que parece estar mas cerca del objetivo

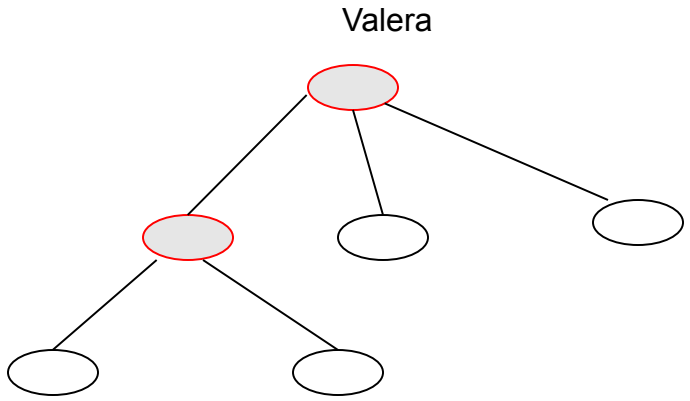
Ejemplo: Venezuela

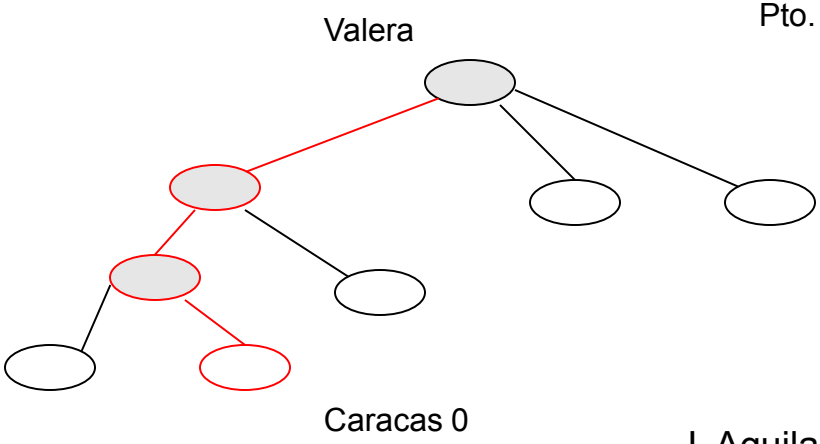


Solución


Valera


Valera
Bqto: 288
Macbo: 549
Barinas: 504


Valera
Pto. Cabello: 197
Valencia: 211


Valera
Caracas 0

Búsqueda con vecindad variable

Se **exploran vecindades distantes** de la solución actual de manera incremental, buscando moverse a partir de la solución actual si se ha obtenido alguna mejora en las vecindades próximas

Se fundamenta en un **algoritmo de búsqueda local** mejorado a partir del cambio dinámico de vecindades, bajo las siguientes premisas:

- Un óptimo local en una determinada vecindad, no necesariamente lo es en otra vecindad. Una solución es *k*-vecina de otra, si entre si existen k cambios con respecto a la estructura que las define.
- Un óptimo global es un óptimo local para todas las estructuras de vecindades. Para muchos problemas de optimización, los óptimos locales están relativamente entre ellos.

Búsqueda con vecindad variable

PASOS

- Agitación (Shaking) + Búsqueda Local
- Movimiento (Move)

```
1  S_o; // solución inicial
2  S_φ; // mejor solución encontrada
3  S; // solución actual
4  FUNCTION VariableNeighbourhoodSearch
5  S ← S_o;
6  S_φ ← S_o;
7  while stopping criterion is not reached do
8  |  k ← 1;
9  |  while k ≤ k_max do
10 |  |  S ←NEIGHBORHOOD(S, k); // La función de vecindad devuelve la
11 |  |                        // mejor solución vecina de S. Donde k
12 |  |                        // indica la k-ésima estructura de
13 |  |                        // vecindad
14 |  |  if f(S) < f(S_φ) then
15 |  |  |  S_φ ← S; // actualiza la mejor solución
16 |  |  |  k ← 1;
17 |  |  else
18 |  |  |  k ← k + 1;
19 |  |  end if
20 |  end while
21 end while
22 return S_φ
```

Metaheurísticas

métodos generales para construir heurísticas para una gran variedad de problemas.

- Recocido Simulado
- Algoritmos Genéticos
- Técnicas bioinspiradas

Temple simulado (iterativo)

- También conocido como recocido simulado, recibe su nombre porque su comportamiento se asemeja al proceso de recocido del acero y del vidrio.
- **Es un algoritmo de búsqueda local**:
  - Parte de una solución inicial seleccionada de manera aleatoria.
  - Seguidamente, un vecino de esta solución es generado por algún mecanismo adecuado y el cambio en el costo es calculado.
  - Si el costo se reduce con respecto a la solución actual, ésta es reemplazada por el vecino generado (dependiendo del valor de la temperatura), de otra manera la solución se mantiene.
  - El proceso se repite hasta que no se encuentran mejoras en la vecindad de la solución actual (mínimo local), ó cuando se llega a cierta temperatura.
- **Una de sus principales desventajas es que el mínimo local puede estar muy lejos del mínimo global**. Para ello, el algoritmo busca evitar caer en el mínimo local aceptando en algunas ocasiones vecinos malos según cierta probabilidad dada por la función de aceptación

Recocido Simulado

Un **esquema de enfriamiento** que regula cómo va disminuyendo gradualmente la temperatura.

$$T(k) = \frac{T_o}{1 + \ln k}$$

Un algoritmo que se utiliza para encontrar la distribución de equilibrio para cada nuevo valor de la temperatura obtenido por el esquema de enfriamiento.

Temple Simulado (iterativo)

- actual =solución_inicial[problema]
- T= alta temperatura
- repita hasta T= congelado
  - próximo= aleatoria selección desde actual de una solución vecina
  - E= costo[próximo]- costo[actual]
  - Si T= congelado y E> 0 entonces
    - regresar actual

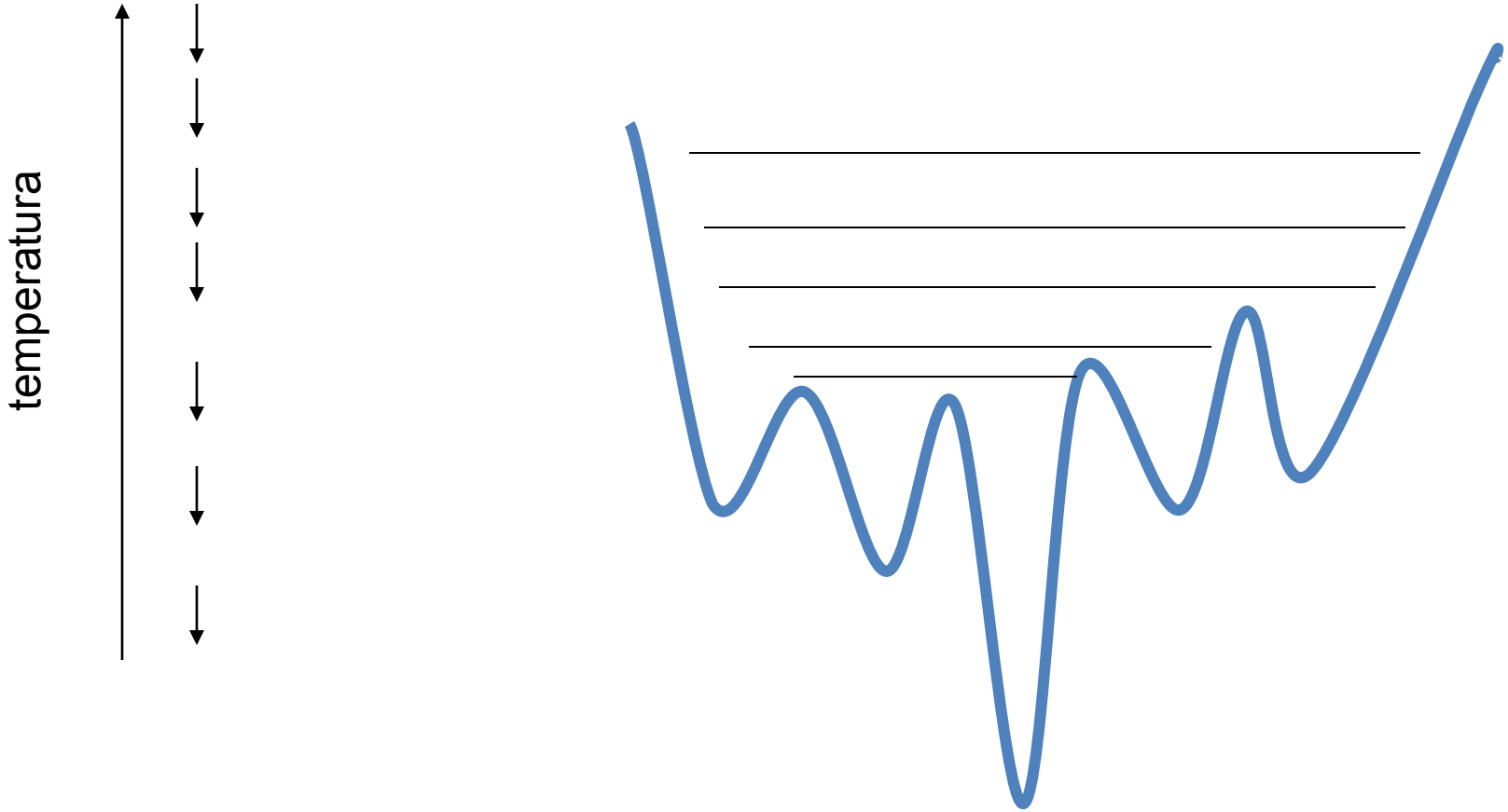
  de lo contrario
    - Si E>0 entonces
      - actual= próximo con probabilidad $e^{E/T}$
    - Si E< 0 entonces
      - actual= próximo
    - descender T si ya han sido aceptados un número dado de movimientos para ese T

parada

Actualiza solución

Enfriamiento (equilibrio)

Búsqueda del Mínimo Global : Recocido Simulado



Búsqueda por profundidad

Se puede retroceder en la busqueda para conseguir solución que satisfaga restricciones (Backtracking)

```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failue then return result
            remove { var = value } from assignment
    return failure
```

Backtracking (vuelta atrás)

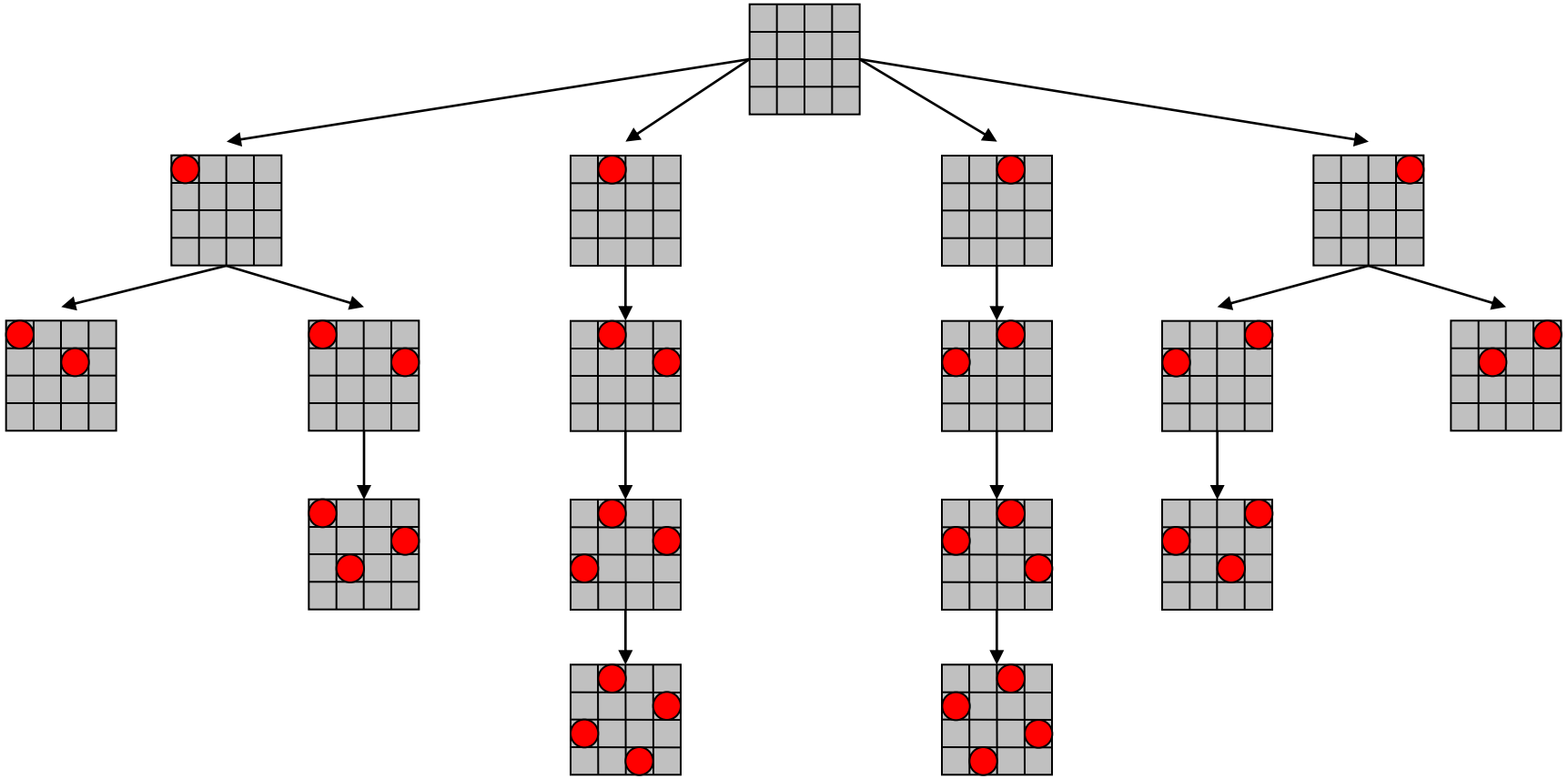
Es un técnica que se utiliza cuando el resolver un problema se traduce a buscar un nodo, un camino o un patrón especifico en el grafo asociado.

Versión 1.0

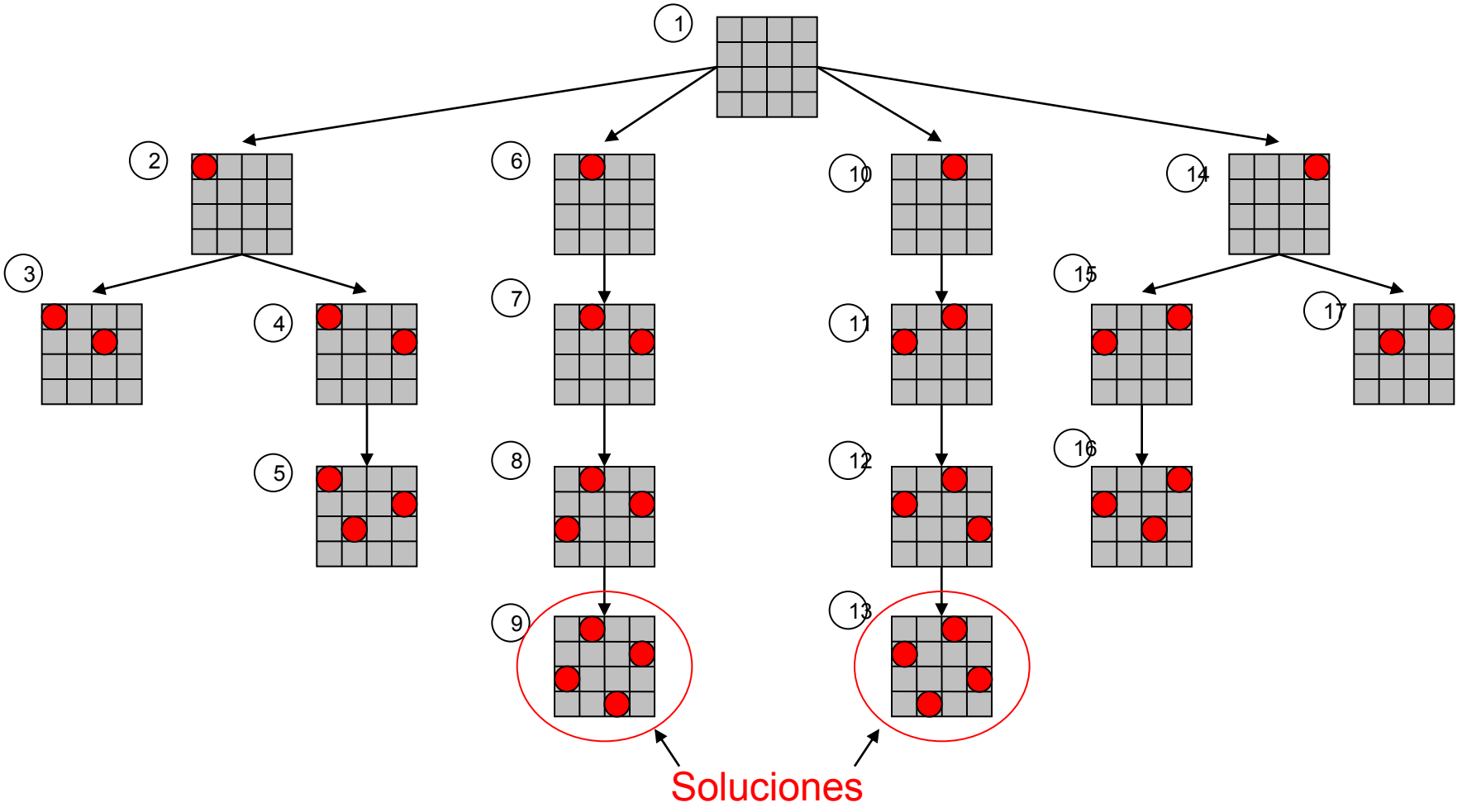
mochilava (Entero: i, r): Entero: b
{Calcula el valor de la mejor carga que se puede construir empleando elementos de los tipos i a n y cuyo peso total no sobrepase r}
{pre: n>0, W>0} **{pos: entero con la mejor carga}**

1	**b=0**	**Para este algoritmo suponemos que los valores n y w, y las arreglos w[1..n] y v[1..n] están disponibles como variables globales.**
2	**[Si (w[k]<= r) entonces** **b=máx(b, v[k] + mochilava(k ,r-w[k]))** **]k=i,n**	
3	**Regrese b**	

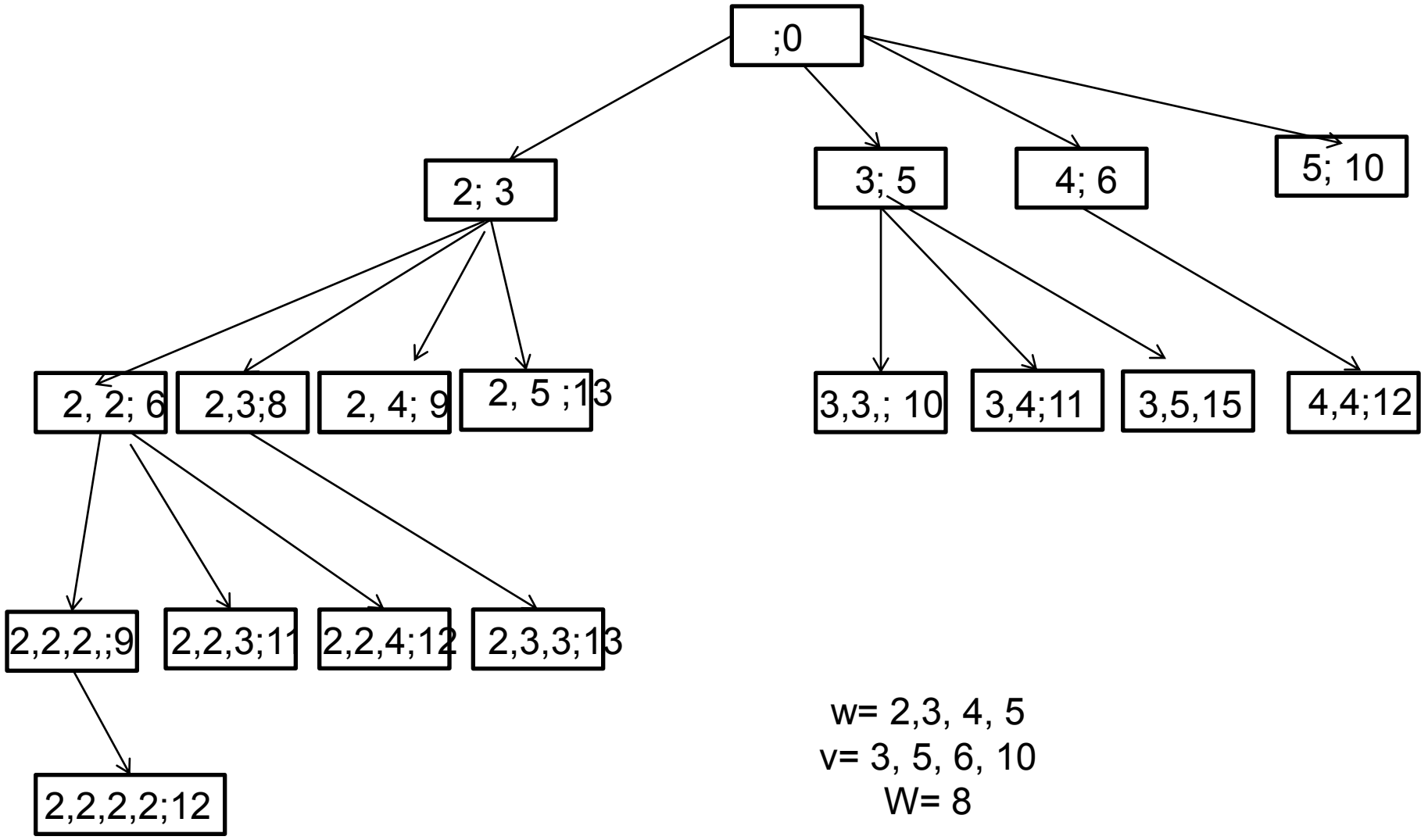
Backtracking en el problema 4-Reinas



Backtracking en el problema 4-Reinas

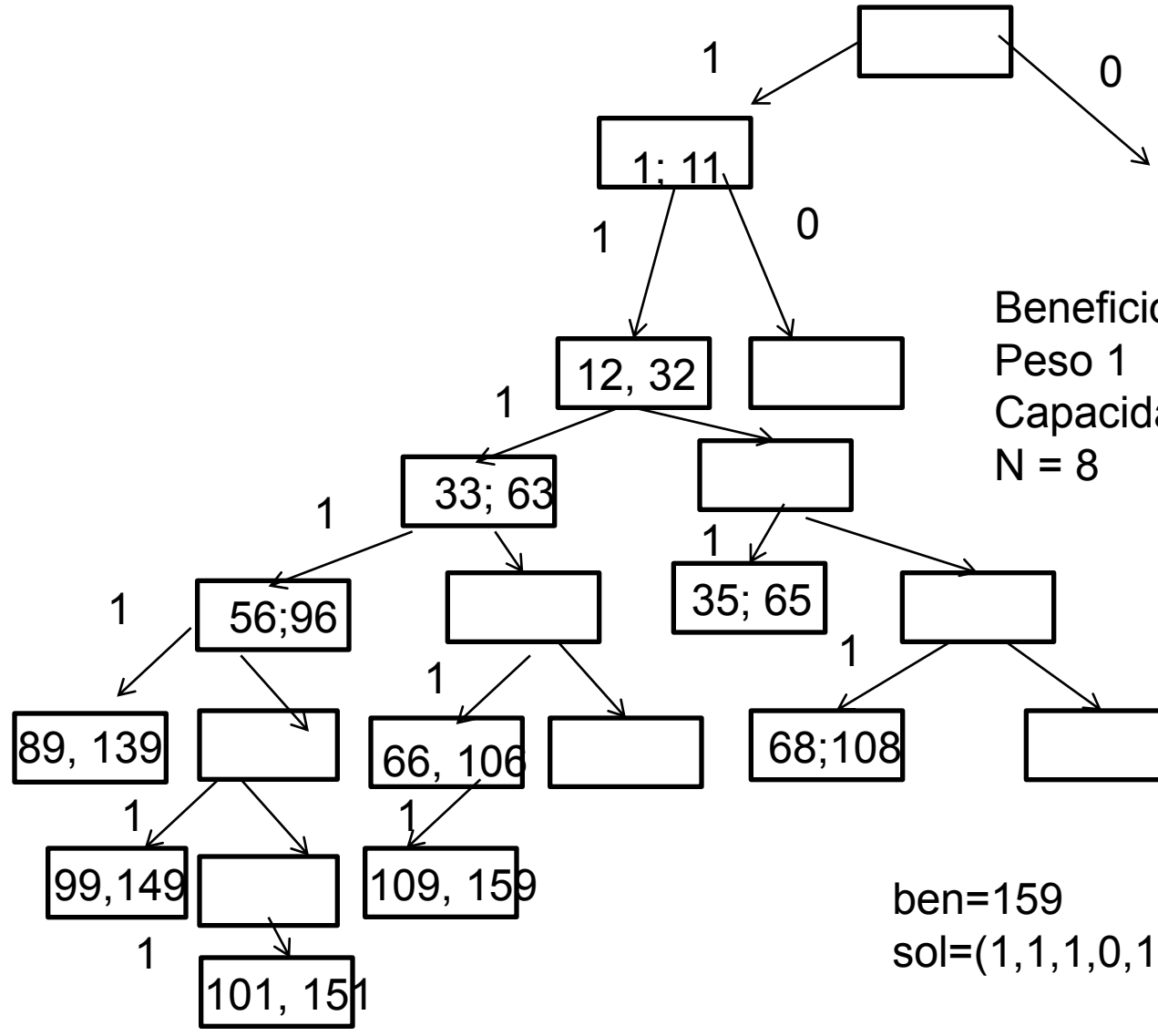


El problema de la mochila empleando vuelta atrás,



w= 2,3, 4, 5
v= 3, 5, 6, 10
W= 8

El problema de la mochila empleando vuelta atrás,



Beneficio 11 21 31 33 43 53 55 65
Peso 1 11 21 23 33 43 45 55
Capacidad = 110
N = 8

ben=159
sol=(1,1,1,0,1,1,0,0

Algoritmos Golosos

- **Idea:** Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar las implicancias de esta selección en los pasos futuros ni en la solución final.

Se usan principalmente para problemas de optimización.

Algoritmos Golosos

Componentes:

- Lista o conjunto de candidatos
- Conjunto de candidatos ya usados
- Función que verifica si un conjunto de candidatos da una solución al problema.
- Función que verifica si un conjunto de candidatos es hasta el momento factible.
- Función de selección
- Función objetivo

Algoritmos Golosos

Función GREEDY (C)

{C es el conjunto de candidatos}

S ←------ ∅

Mientras NOT solución(S) y C ≠ ∅ hacer

x ←---------- elemento que maximice select (x)

C <---------- C \ {x}

Si S U {x} es factible entonces s ←--------- S U {x}

Si solución(S) entonces RETURN S

sino RETURN "no hay solución"

Minimizar el tiempo de espera en un sistema

Un servidor tiene n clientes para atender. Se sabe que el tiempo requerido para atender cada cliente i es t_i. Se quiere minimizar

$$T = \Sigma_i \text{ (tiempo que i está en el sistema)}$$

El algoritmo goloso que atiende al cliente que requiere el menor tiempo de atención entre los que aún están en la cola resuelve el problema.

Búsqueda Tabú

- Clasificación:
  - Es determinista (vs. aleatoria)
  - Es basada en un individuo (vs. poblaciones)
  - Es iterativo.
    - Hay que definir la vecindad utilizada.
  - Utiliza memoria.

- Combina ***búsqueda local con una heurística*** para evitar parar en mínimos locales y evitar entrar en ciclos.

- **Algoritmo**

  1. Elegir una solución inicial i en S
  2. Generar un subconjunto V* de N(i)
  3. Elegir el mejor j en V* (es decir tales que f(j)<=f(k) para cualquier k en V *)
  4. Si f(j) >= f(i) entonces terminar. Si no, establecer i=j e ir al paso 2

- Para V* = N(i) tenemos el caso de la búsqueda local.

Algoritmos Tabú

- **Otro algoritmo**

  1. Elegir una solución inicial i en S. Establecer i*=i.
  2. Agregar i a la lista tabú.
  3. Elegir el mejor j en V* = N(i) \ ListaTabu.
  4. Si f(i) < f(i*) asignar i*=i.
  5. Si se cumple alguna condición de fin, terminar. Si no asignar i=j e ir al paso 2.

- El caso de búsqueda local es el caso particular en que la condición de parada es f(i)>=f(i*)

Búsqueda en profundidad Branch-and-Bound (Ramificación y Poda)

- Técnica similar a Backtracking
- Al igual que en backtracking, el algoritmo realiza una búsqueda sistemática en un árbol de soluciones.
- Continúa buscando después de encontrar una solución, de manera que actualiza la mejor solución.
- Poda los caminos solución que conducen a una peor solución.
- Posible poda de nodos que mejoran el camino solución.

Ramificación y Poda

Puede ser visto como una generalización o mejora del BACKTRACKING.

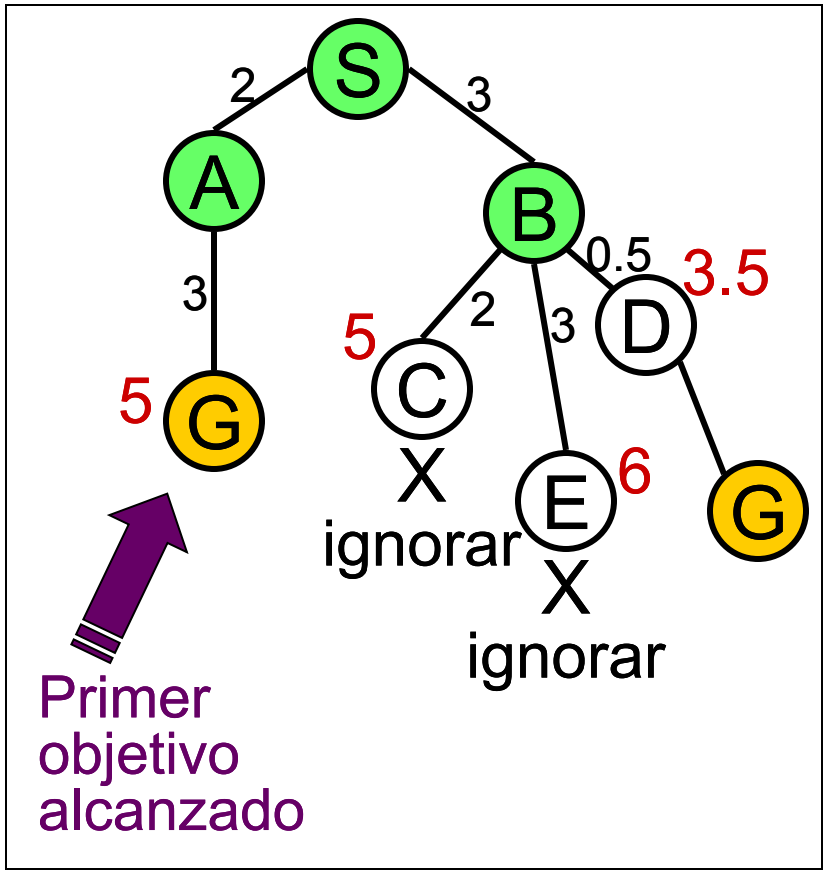
- En Bactracking, en cuanto se genera un nuevo nodo hijo pasa a procesarse
- En Ramificación y Poda, se generan todos los nodos hijos del nodo actual y después se van procesando
- En Backtracking los únicos nodos vivos son los que están en el camino desde la raíz hasta el nodo que se está estudiando
- En Ramificación y Poda puede haber más nodos vivos que se almacenan en la lista de nodos vivos

Búsqueda en profundidad Branch-and-Bound (Ramificación y Poda)

- **Añade dos características nuevas**:
  - Estrategia de ramificación: La búsqueda se guiará por estimaciones de beneficio que se harán en cada nodo.
  - Estrategia de poda: Para eliminar nodos que no lleven a a la solución óptima. Estimación de cotas de beneficio en cada nodo.

- **Para cada nodo tendremos la siguiente información:**
  - Cota inferior (CI) y cota superior (CS) de beneficio que se puede obtener a partir del nodo (para realizar una poda).
  - Estimación del beneficio que se puede encontrar a partir del nodo. Ayuda a decidir el orden de evaluación de los nodos.

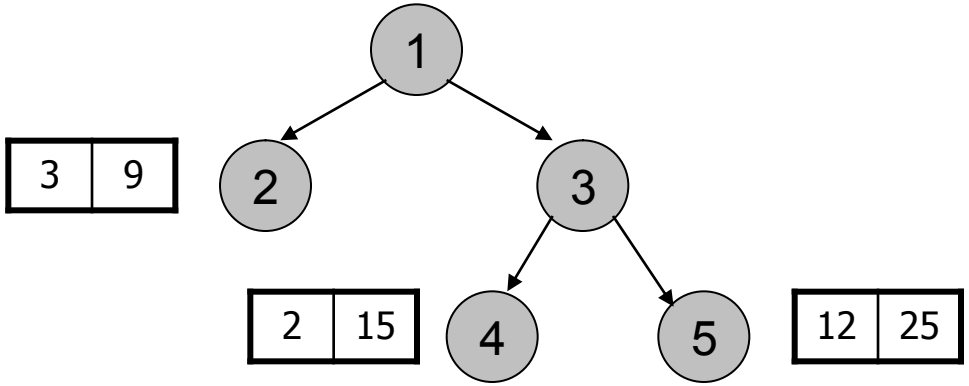
Principio Búsqueda en profundidad Branch-and-Bound



- Usar cualquier método de búsqueda (completo) para encontrar una solución (camino).
- Remover todos los caminos parciales que tengan un costo acumulado mayor o igual que el camino hallado.
- Continuar la búsqueda para el próximo camino.

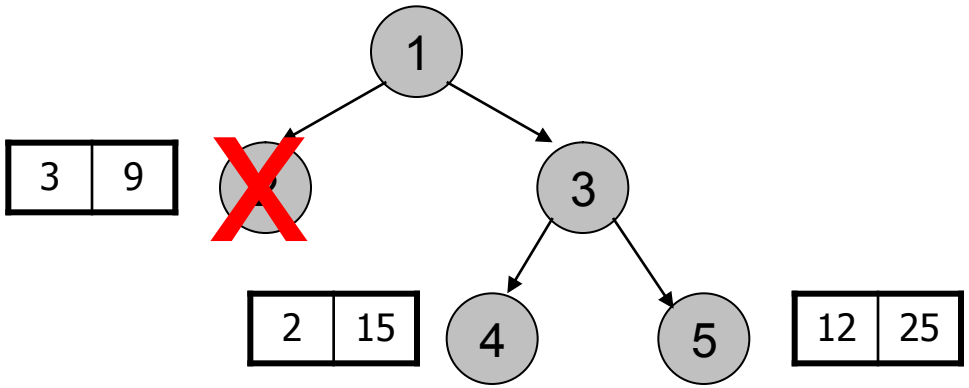
Estrategia de poda

- Suponemos un problema de maximización.
- Se han recorrido varios nodos y ha estimado la cota inferior y superior de cada uno de ellos.



Estrategia de poda

- El nodo 2 puede podarse, ya que el nodo 5 tiene una cota inferior mayor que la cota superior del nodo 2.



Estrategia de ramificación

- Se pueden utilizar distintas estrategias para recorrer el árbol de soluciones. Puede recorrerse en profundidad, en anchura, según el beneficio estimado,…

- Se utiliza una lista de nodos vivos (contiene nodos generados y aún no explorados).

- Algoritmo:
  - Sacar un nodo de la lista de nodos vivos
  - Generar sus descendientes
  - Si no se podan, se introducen en la lista de nodos vivos.

Estrategia de ramificación

- ¿Qué criterio se utiliza para seleccionar el nodo que se saca de la lista de nodos vivos?
  - Estrategia FIFO
  - Estrategia LIFO
  - Estrategia del menor costo

UNIVERSIDAD DE LOS ANDES
MERIDA VENEZUELA

Ramificación y Poda

Versión 1.0

ramificacionYpoda(Tipo_Nodo: raiz, Tipo_Nodo: s)

1	LNV = {raiz} C = CotaSuperior(raiz) S = {Ø}	--LNV: lista de nodos vivos -CotaSuperior():Cota superior del beneficio óptimo -seleccionar(): Método de selección del prox nodo a probar. -solucion():Comprobar si es una solución final y tratarla

2	(LNV ≠ {Ø})[
	X = seleccionar(LNV) // estrategia
	//Ramificación
	LNV = LNV – {X}
	si (CotaInferior(X) < C) entonces //PODA
	Para Cada Hijo Y de X Hacer
	SI (solucion (Y) y Valor (Y) < valor (S))
	S = Y
	C = min(C, Valor (Y))
	sino
	si (! Solucion (Y) y CotaInferior(Y) < C)
	LNV = LNV + {Y}
	C = min(C, CotaSuperior(Y))
	FinSi
	FinSi
	FinPara	