Teoría de Algoritmos

Capitulo 5: Algoritmos para la Exploración de Grafos.

Tema 14: Backtracking y Branch and Bound

- Branch and Bound
  - Problema de la Mochila
  - Problema del Viajante de Comercio
  - Los 15 numeros

Branch and bound

- **Branch and Bound** es una técnica muy similar a la de **Backtacking**, y basa su diseño en el análisis del árbol de estados de un problema:
  - Realiza un **recorrido sistemático** de ese árbol.
  - El recorrido no tiene que ser necesariamente en profundidad.
- Generalmente se aplica para resolver problemas de Optimización (Programación Matemática) y para jugar Juegos

Branch and bound

- Los algoritmos generados por está técnica son normalmente de orden **exponencial** o peor en su peor caso, pero su aplicación en casos muy grandes, ha demostrado ser eficiente (incluso más que bactracking).
- Puede ser vista como una **generalización** (o mejora) de la técnica de Backtracking.
- Tendremos una **estrategia de ramificación**.
- Se tratará como un aspecto importante las **técnicas de poda**, para eliminar nodos que no lleven a soluciones optimas.
- La poda se realiza **estimando** en cada nodo **cotas** del beneficio que podemos obtener a partir del mismo.

Branch and bound

- Diferencia fundamental con Backtracking:
  - En Bactracking tan pronto como se genera un nuevo hijo del nodo en curso, este hijo pasa a ser el **nodo en curso**.
  - En BB se generan todos los hijos del nodo en curso antes de que **cualquier otro nodo vivo** pase a ser el nuevo nodo en curso (esta técnica no utiliza la búsqueda en profundidad)
- En consecuencia:
  - En Backtracking los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso.
  - En BB puede haber más nodos vivos (se almacenan en una estructura de datos auxiliar: **lista de nodos vivos**.
- Además
  - En Bactracking el test de comprobación nos decía si era fracaso o no, mientras que en BB la cota nos sirve para podar el árbol y para saber el orden de ramificación, comenzando por las más prometedoras

Descripción General del Método

- BB es un método de búsqueda general que se aplica conforme a lo siguiente:
- Explora un árbol comenzando a partir de un problema raiz (el problema original con su región factible completa)
- Entonces se aplican procedimientos de cotas inferiores y superiores al problema raiz.
- Si las cotas cumplen las condiciones que se hayan establecido, habremos encontrado la solución optimal y el procedimiento termina.
- Si ese no fuera el caso, entonces la región factible se divide en dos o mas regiones, dando lugar a distintos subproblemas.
- Esos subproblemas particionan la región factible. La búsqueda se desarrolla en cada una de esas regiones.
- El método (algoritmo) se aplica recursivamente a los subproblemas.

Descripción General del Método

- Si se encuentra una solución optimal para un subproblema, será una solución factible para el problema completo, pero no necesariamente el optimo global.
- Cuando en un nodo (subproblema) la cota superior es menor que el mejor valor conocido en la región, no puede existir un optimo global en el subespacio de la región factible asociada a ese nodo, y por tanto ese nodo puede ser eliminado en posteriores consideraciones.
- La búsqueda sigue hasta que se examinan o "podan" todos los nodos, o hasta que se alcanza algun criterio pre-establecido sobre el mejor valor encontrado y las cotas superiores de los problemas no resueltos

Branch and bound

- **Para cada nodo i tendremos:**
  - **Cota superior** (**CS(i)**) y **Cota inferior** (**CI(i)**) del beneficio (o coste) óptimo que podemos alcanzar a partir de ese nodo.
    - Determinan cuándo se puede realizar una poda.
  - **Estimación del beneficio (o coste)** óptimo que se puede encontrar a partir de ese nodo. Puede ser una media de las anteriores.
    - Ayuda a decidir qué parte del árbol evaluar primero.
  - **Estrategia de poda**
    - Suponemos un problema de maximización
    - Hemos recorrido varios nodos 1..n, estimando para cada uno la cota superior **CS (j)** e inferior **CI (j)**, respectivamente, para **j** entre 1 y n.
    - Hay dos casos

Branch and bound

- ***CASO 1.*** Si a partir de un nodo siempre podemos obtener alguna solución válida, entonces **podar un nodo i si**:

  CS (i) ≤ CI (j), para algún nodo j generado.

- **Ejemplo:** problema de la mochila, utilizando un árbol binario.
  - A partir de **a**, podemos encontrar un beneficio máximo de CS(a)=4.
  - A partir de **b**, tenemos garantizado un beneficio mínimo de CI(b)=5.
  - Podemos podar el nodo **a**, sin perder ninguna solución óptima.

- ***CASO 2.*** Si a partir de un nodo puede que no lleguemos a una solución válida, entonces podemos **podar un nodo i si**:

  CS (i) ≤ Beneficio (j), para algún j, solución final (factible).

- **Ejemplo:** problema de las reinas. A partir de una solución parcial, no está garantizado que exista una solución

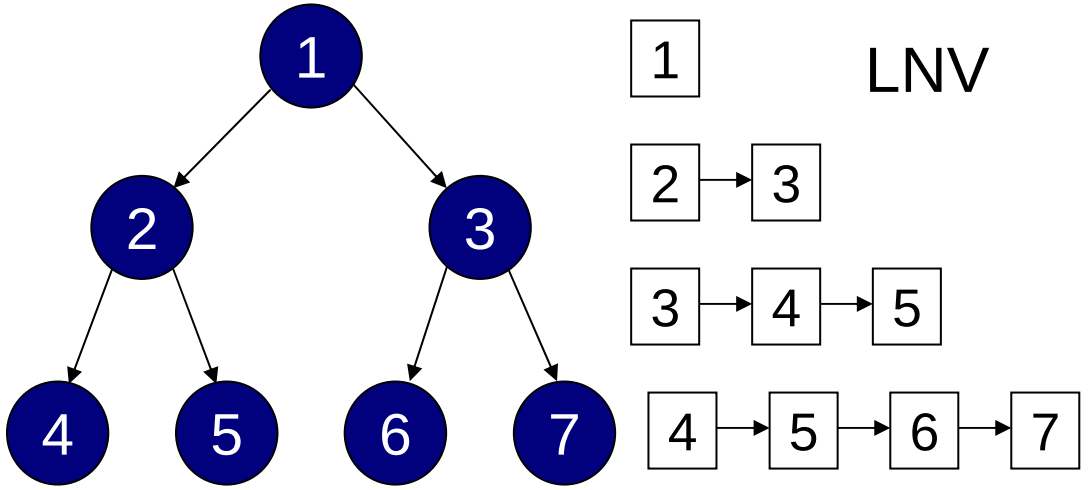
Branch and bound

Estrategias de ramificación

- Normalmente el árbol de soluciones es implícito, no se almacena en ningún lugar.
- Para hacer el recorrido se utiliza una **lista de nodos vivos**.
- **Lista de nodos vivos:** contiene todos los nodos que han sido generados pero que no han sido explorados todavía. Son los nodos pendientes de tratar por el algoritmo.
- Según cómo sea la lista, el recorrido será de uno u otro tipo.

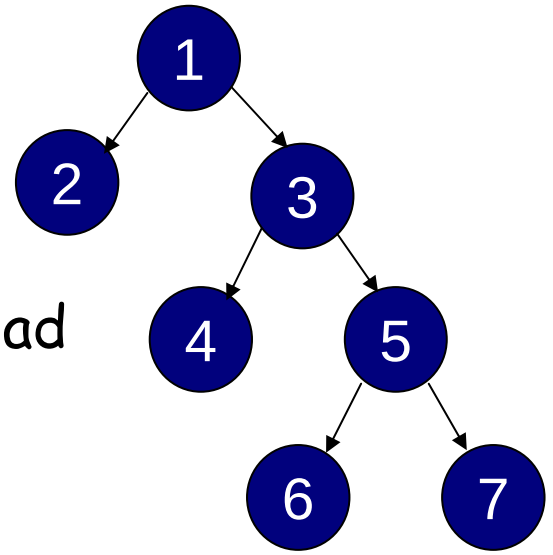
Estrategia FIFO (First In First Out)

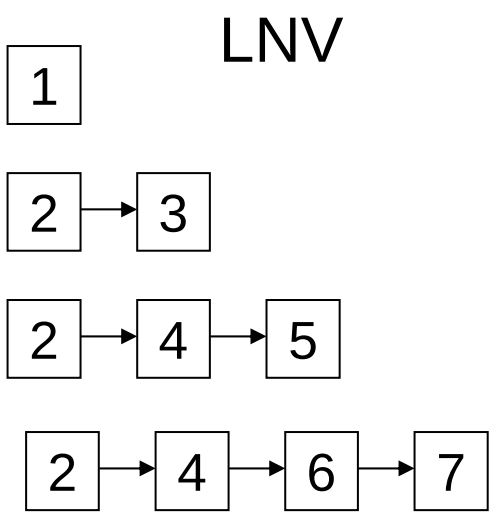
La lista de nodos vivos es una cola
El recorrido es en anchura



Branch and bound

ESTRATEGIA LIFO
(Last In First Out)
La lista de nodos vivos
es una pila
El recorrido es en profundidad





- Las estrategias FIFO y LIFO realizan una búsqueda "a ciegas", sin tener en cuenta los beneficios.
- Usando la estimación del beneficio, entonces será mejor buscar primero por los nodos con mayor valor estimado.
- **Estrategias LC (Least Cost):**
  - Entre todos los nodos de la lista de nodos vivos, elegir el que tenga mayor beneficio (o menor coste) para explorar a continuación.

Branch and bound

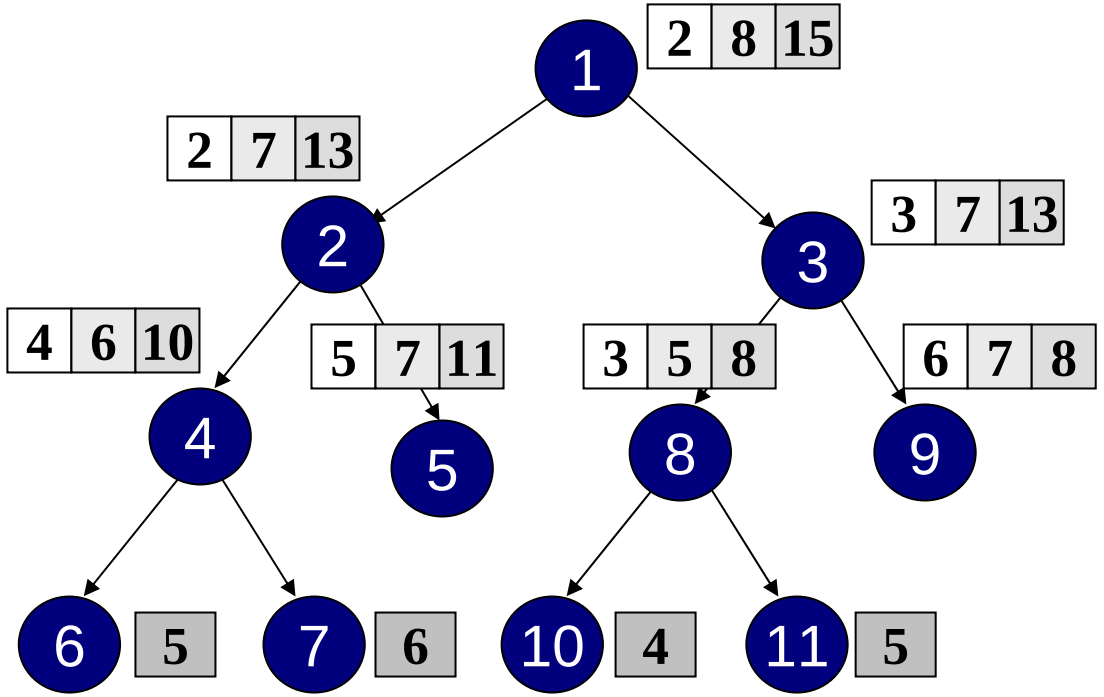
Estrategias de ramificación LC

- En caso de empate (de beneficio o coste estimado) deshacerlo usando un criterio **FIFO** ó **LIFO**:
  - **Estrategia LC-FIFO:** Seleccionar de la LNV el que tenga mayor beneficio y en caso de empate escoger el primero que se introdujo (de los que empatan).
  - **Estrategia LC-LIFO:** Seleccionar de la LNV el que tenga mayor beneficio y en caso de empate escoger el último que se introdujo (de los que empatan).

- En cada nodo podemos tener: cota inferior de coste, coste estimado y cota superior del coste.
- Podar según los valores de las cotas.
- Ramificar según los costes estimados.

BB: El Método General

- **Ejemplo. Branch and Bound usando LC-FIFO.**
- Supongamos un problema de minimización, y que tenemos el caso 1 (a partir de un nodo siempre existe alguna solución).
- Para realizar la poda usaremos una variable ***C*** = valor de la menor de las cotas superiores hasta ese momento (o de alguna solución final).
- Si para algún nodo i, $CI(i) \geq C$, entonces podar i.



C	LNV
15	1
13	2 → 3
10	4 → 3 → 5
5	3 → 5
5	8 → 5
4	5

BB: El Método General

- **Esquema general**. Problema de minimización, suponiendo el caso en que existe solución a partir de cualquier nodo.

```
RamificacionYPoda (NodoRaiz: tipo_nodo; var s: tipo_solucion);
    LNV:= {NodoRaiz};
    C:= CS (NodoRaiz);
    s:= ∅;
    Mientras LNV ≠ ∅ hacer
          x:= Seleccionar (LNV);  { Según un criterio FIFO, LIFO, LC-FIFO ó
    LC-LIFO }
          LNV:= LNV - {x};
          Si CI (x) < C entonces                    { Si no se cumple se poda x }
               Para cada y hijo de x hacer
                    Si y es una solución final mejor que s entonces
                         s:= y;
                         C:= min (C, Coste (y) );
                    Sino si y no es solución final y (CI(y) < C) entonces
                         LNV:= LNV + {y};
                         C:= min (C, CS (y) );
               FinPara;
    FinMientras;
```

BB: Análisis de los tiempos de ejecución

- El tiempo de ejecución depende de:
  - **Número de nodos recorridos**: depende de la efectividad de la poda.
  - **Tiempo gastado en cada nodo**: tiempo de hacer las estimaciones de coste y tiempo de manejo de la lista de nodos vivos.
- En el peor caso, el tiempo es igual que el de un algoritmo con backtracking (ó peor si tenemos en cuenta el tiempo que requiere la LNV).
- En el caso promedio se suelen obtener mejoras respecto al backtracking.
- ¿Cómo hacer que un algoritmo BB sea más eficiente?
  - **Hacer estimaciones de costo muy precisas**: Se realiza una poda exhaustiva del árbol. Se recorren menos nodos pero se gasta mucho tiempo en realizar las estimaciones.
  - **Hacer estimaciones de costo poco precisas**: Se gasta poco tiempo en cada nodo, pero el número de nodos puede ser muy elevado. No se hace mucha poda.
- Se debe buscar un equilibrio entre la exactitud de las cotas y el tiempo de calcularlas.

Ejemplo, El Problema de la Mochila 0/1

- **Diseño del algoritmo** BB:
  - Definir una representación de la solución. A partir de un nodo, cómo se obtienen sus descendientes.
  - Dar una manera de calcular el valor de las cotas y la estimación del beneficio.
  - Definir la estrategia de ramificación y de poda.

- **Representación de la solución:**
  - **Mediante un árbol binario**: $(s_1, s_2, ..., s_n)$, con $s_i = (0, 1)$.
    Hijos de un nodo $(s_1, s_2, ..., s_k)$: $(s_1, ..., s_k, 0)$ y $(s_1, ..., s_k, 1)$.
  - **Mediante un árbol combinatorio**: $(s_1, s_2, ..., s_m)$ donde $m \leq n$ y $s_i \in \{1, 2, ..., n\}$.
    Hijos de un nodo $(s_1, .., s_k)$:
    $(s_1, .., s_k, s_k+1), (s_1, ..., s_k, s_k+2), ..., (s_1, ..., s_k, n)$

Ejemplo, El Problema de la Mochila 0/1

- **Cálculo de cotas:**
  - **Cota inferior**: Beneficio que se obtendría incluyendo sólo los objetos incluidos hasta ese nodo.
  - **Estimación del beneficio**: A la solución actual, sumar el beneficio de incluir los objetos enteros que quepan, utilizando avance rápido. Suponemos que los objetos están ordenados por orden decreciente de v_i/w_i.
  - **Cota superior**: Resolver el problema de la mochila continuo a partir de ese nodo (con un algoritmo greedy), y quedarse con la parte entera.

- **Ejemplo**. n = 4, M = 7, v = (2, 3, 4, 5), w = (1, 2, 3, 4)

  Nodo actual: (1, 1) (1, 2)
  Hijos: (1, 1, 0), (1, 1, 1) (1, 2, 3), (1, 2, 4)
  Cota inferior: $CI = v_1 + v_2 = 2 + 3 = 5$
  Estimación del beneficio: $EB = CI + v_3 = 5 + 4 = 9$
  Cota superior: $CS = CI + \lfloor MochilaGreedy(3, 4) \rfloor = 5 + \lfloor 4+5/4 \rfloor = 10$

Ejemplo, El Problema de la Mochila 0/1

- **Forma de realizar la poda:**
  - En una variable ***C*** guardar el valor de la mayor cota inferior hasta ese momento dado.
  - Si para un nodo, su cota superior es menor o igual que ***C*** entonces se puede podar ese nodo.
- **Estrategia de ramificación:**
  - Puesto que tenemos una estimación del coste, usar una estrategia **LC**: explorar primero las ramas con mayor valor esperado **(MB)**.
  - ¿LC-FIFO ó LC-LIFO? Usaremos la LC-LIFO: en caso de empate seguir por la rama más profunda. (MB-LIFO)

Ejemplo, El Problema de la Mochila 0/1

```
MochilaO1RyP (v, w: array [1..n] of integer; M: integer; var s: nodo);
    inic:= NodoInicial (v, w, M);
    C:= inic.CI;
    LNV:= {inic};
    s.v_act:= -∞;
    Mientras LNV ≠ ∅ hacer
          x:= Seleccionar (LNV);          { Segun el criterio MB-LIFO }
          LNV:= LNV - {x};
          Si  x.CS > C  Entonces          { Si no se cumple se poda x }
               Para  i:= 0, 1  Hacer
                    y:= Generar (x, i, v, w, M);
                    Si (y.nivel = n)  Y  (y.v_act > s.v_act)  Entonces
                          s:= y;
                          C:= max (C, s.v_act );
                    Sino Si  (y.nivel < n)  Y  (y.CS > C)  Entonces
                          LNV:= LNV + {y};
                          C:= max (C, y.CI );
               FinPara;
          FinSi;
    FinMientras;
```

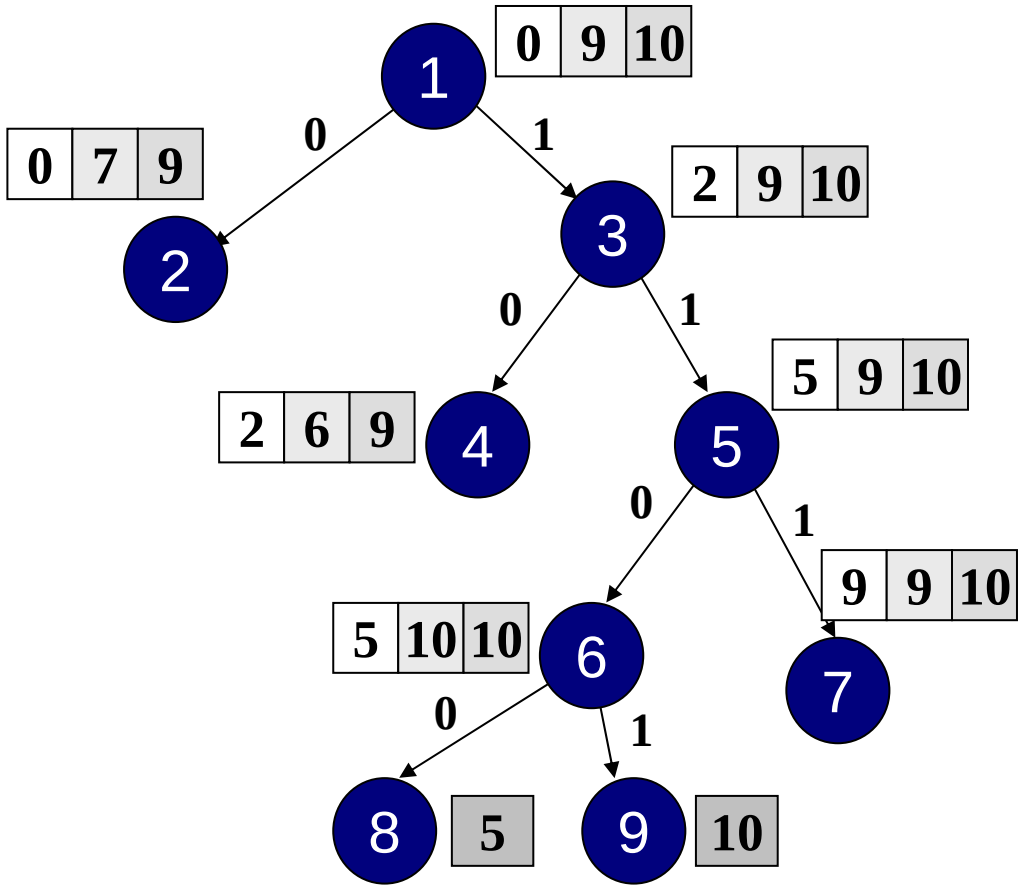
Ejemplo, El Problema de la Mochila 0/1

```
NodoInicial (v, w: array [1..n] of integer; M: integer) : nodo;
    res.CI:= 0;
    res.CS:= ⌊MochilaVoraz (1, M, v, w)⌋;
    res.BE:= Mochila01Voraz (1, M, v, w);
    res.nivel:= 0;
    res.v_act:= 0;          res.w_act:= 0;
    Devolver res;

Generar (x: nodo; i: (0, 1); v,w: array [1..n] of int; M: int): nodo;
    res.tupla:= x.tupla;
    res.nivel:= x.nivel + 1;
    res.tupla[res.nivel]:= i;
    Si  i = 0  Entonces  res.v_act:= x.v_act;  res.w_act:= x.w_act;
    Sino  res.v_act:= x.v_act + v[res.nivel]; res.w_act:= x.w_act +
    w[res.nivel];
    res.CI:= res.v_act;
    res.BE:= res.CI + Mochila01Voraz (res.nivel+1, M - res.w_act, v, w);
    res.CS:= res.CI + ⌊MochilaVoraz (res.nivel+1, M - res.w_act, v, w)⌋;
    Si res.w_act > M Entonces {Sobrepasa el peso M: descartar el nodo }
          res.CI:= res.CS:= res.BE:= -∞;
    Devolver res;
```

Ejemplo, El Problema de la Mochila 0/1

- **Ejemplo**. n = 4, M = 7, v = (2, 3, 4, 5), w = (1, 2, 3, 4)



C	LNV
0	**1**
2	**3** → 2
5	**5** → 2 → 4
9	**6** → 7 → 2 → 4
10	**7** → 2 → 4
10	**2** → 4
10	**4**

El Problema del Viajante de Comercio

- Este problema fue resuelto con Programación dinámica, obteniendo un algoritmo de orden $O(n^2 2^n)$...
- Para un 'n' grande, el algoritmo es ineficiente...
- Branch and bound se adapta para solucionarlo...

El Problema del Viajante de Comercio

- Formalización:
  - Sean $G = (V,A)$ un grafo orientado, $V=\{1,2,\ldots,n\}$,
  - $D[i,j]$ la longitud de $(i,j) \in A$, $D[i,j]=\infty$ si no existe el arco (i,j).
  - El circuito buscado empieza en el vértice 1.

  - Candidatos:
    $E = \{ 1,X,1 \mid X$ es una permutación de $(2,3,\ldots,n) \}$
    $|E| = (n-1)!$

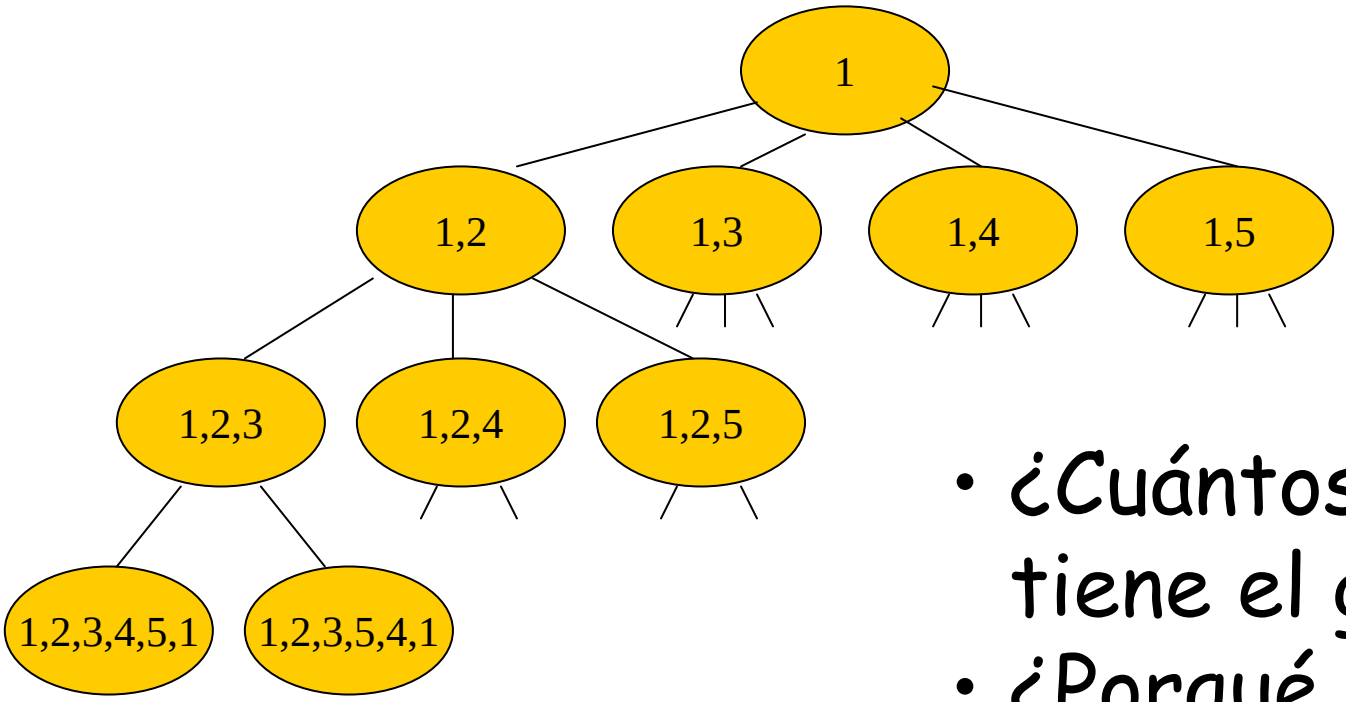
  - Soluciones factibles:
    $E = \{ 1,X,1 \mid X = x_1,x_2,\ldots,x_{n-1},$ es una permutación de $(2,3,\ldots,n)$ tal que $(i_j,i_{j+1}) \in A$, $0<j<n$, $(1, x_1) \in A$, $(x_{n-1},1) \in A\}$

  - Funcion objetivo:
    $F(X)=D[1,x_1]+D[x_1, x_2] + D[x_2, x_3]+\ldots+D[x_{n-2}, x_{n-1}]+D[x_n,1]$

El Problema del Viajante de Comercio

- **Recordatorio**: Ciclo en el grafo en el que TODOS los vértices del grafo se visitan sólo una vez al menor costo.
- Arbol de búsqueda de soluciones:
  - La raíz del árbol (nivel 0) es el vértice de inicio del ciclo.
  - En el nivel 1 se consideran TODOS los vértices menos el inicial.
  - En el nivel 2 se consideran TODOS los vértices menos los 2 que ya fueron visitados.
  - Y así sucesivamente hasta el nivel 'n-1' que incluirá al vértice que no ha sido visitado.

Ejemplo



- ¿Cuántos vértices tiene el grafo?
- ¿Porqué no se requiere el último nivel en el árbol?

Análisis del problema con Branch and Bound

- Criterio de selección para expandir un nodo del árbol de búsqueda de soluciones:
  - Un vértice en el nivel i del árbol, debe ser adyacente al vértice en el nivel i-1 del camino correspondiente en el árbol.
  - Puesto que es un problema de Minimización, si el costo posible a acumular al expandir el nodo i, es **menor** al mejor costo acumulado hasta ese momento, vale la pena expandir el nodo, si no, el camino se deja de explorar ahi...

Estimación del costo posible a acumular

- Si se sabe cuáles son los vértices que faltan por visitar…
- Cada vértice faltante, tiene arcos de salida hacia otros vértices…
- El mejor costo, será el del arco que tenga el valor menor…
- Esta información se puede obtener del renglón correspondiente al vértice en la matriz de adyacencias (excluyendo a los valores cero)…
- La sumatoria de los mejores arcos de cada vértice faltante, más el costo del camino ya acumulado, es una estimación válida para tomar decisiones respecto a las podas en el árbol..

Ejemplo

- Dada la siguiente matriz de adyacencias, ¿cuál es el costo mínimo posible de visitar todos los nodos una sola vez?

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

→ Mínimo = *4*

→ Mínimo = *7*

→ Mínimo = *4*

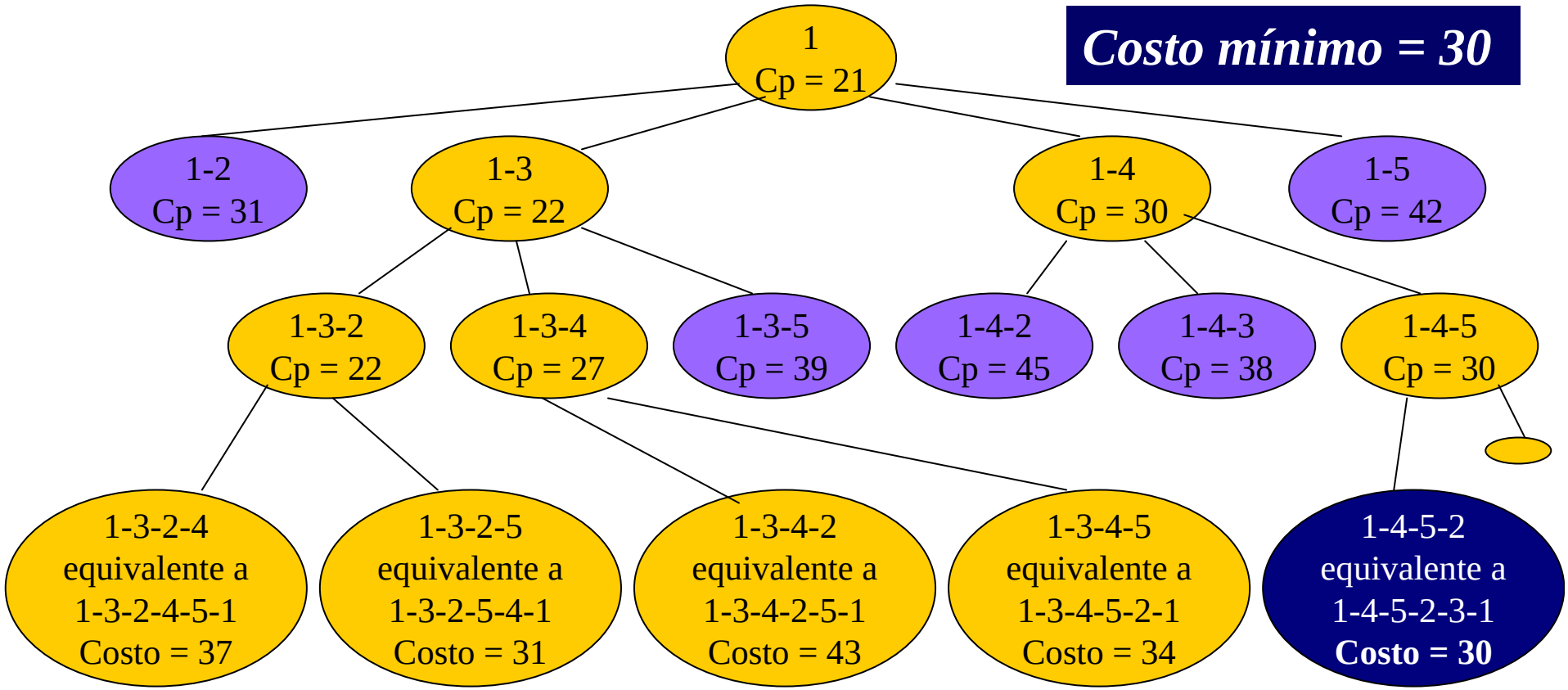
→ Mínimo = *2*

→ Mínimo = *4*

TOTAL = 21

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Conclusión final
El Problema del Viajante de Comercio

- Branch and bound ofrece una opción más de solución del problema del Viajante de Comercio…
- Sin embargo, NO asegura tener un buen comportamiento en cuanto a eficiencia, ya que en el peor caso tiene un tiempo exponencial…
- El problema puede ser resuelto con algoritmos heurísticos: SA, AG, FANS, TS, …

El juego del 15

- **Problema muy difícil para backtracking**
  - **El árbol de búsqueda es potencialmente muy profundo (16! niveles), aunque puede haber soluciones muy cerca de la raíz.**
- **Se puede resolver con branch and bound (aunque hay métodos mejores): *Algoritmo A****
- **funciones de prioridad:**
  - numero de fichas mal colocadas (puede engañar)
  - suma, para cada ficha, de la distancia a la posición donde le tocaría estar
  - El 8-puzzle, introducción a la I.A. clásica