Algorítmica

Tema 5. Algoritmos para la exploración de grafos

Francisco Javier Cabrerizo Lorite

Curso académico 2023-2024

ETS de Ingenierías Informática y de Telecomunicación. Universidad de Granada

Índice

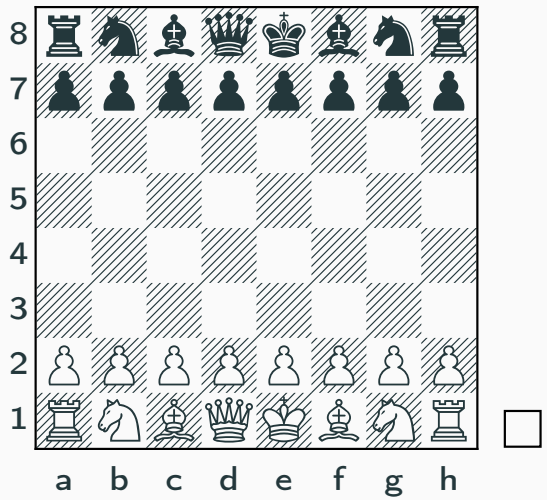
Objetivos

- Ser capaz de proponer diferentes soluciones para un determinado problema y evaluar la calidad de estas.
- Saber ver al árbol de estados como una representación lógica del conjunto de todas las posibles soluciones de un problema.
- Conocer las técnicas de exploración de grafos (vueltas atrás y ramificación y poda) y su aplicación en la resolución de problemas, comprendiendo sus características principales y las diferencias entre ellas.
- Entender y saber aplicar el uso de cotas para reducir el espacio de búsqueda en las técnicas de exploración de grafos.
- Conocer los criterios de aplicación de cada una de las distintas técnicas de diseño de algoritmos.

Introducción

Introducción

- Supongamos que debemos tomar un conjunto de decisiones entre varias alternativas donde:
  - No tenemos suficiente información sobre qué decisión tomar.
  - Cada decisión abre un abanico nuevo de alternativas.
  - Alguna secuencia de decisiones (incluso más de una) puede ser una solución del problema.



Características de los problemas

- La solución debe poder expresarse mediante una tupla

$$(x_1, x_2, \ldots, x_n)$$

  donde cada x_i se selecciona de un conjunto finito S_i.
- La solución consiste en encontrar:
  - La tupla que optimiza (maximiza o minimiza) una función objetivo o criterio $P(x_1, x_2, \ldots, x_n)$.
  - Una tupla que satisfaga (no optimice) el criterio.
  - Todas las tuplas que satisfagan el criterio.

Introducción

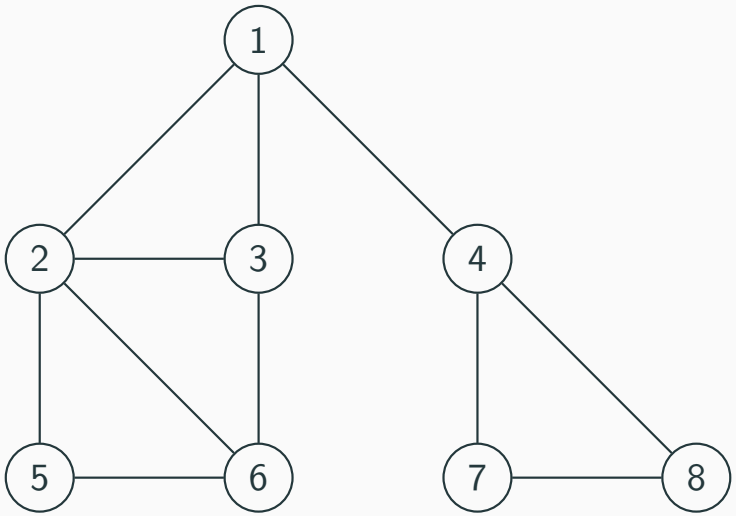
- Muchos de estos problemas se pueden representar mediante un grafo.
- Es necesario conocer las técnicas para su exploración:
  - Recorrido en profundidad.
  - Recorrido en anchura.

Recorrido en profundidad

```
procedure RecorridoProfundidad(G = (V, A))
    for all v ∈ V do
        marca[v] ← no visitado
    end for
    for all v ∈ V do
        if marca[v] = no visitado then
            rp(v)
        end if
    end for
end procedure
```

```
procedure rp(vértice v)
    marca[v] ← visitado
    for all w adyacente a v do
        if marca[w] = no visitado then
            rp(w)
        end if
    end for
end procedure
```

Recorrido en profundidad



`rp(1)`. Se visita el vértice 1 y pasa al adyacente.

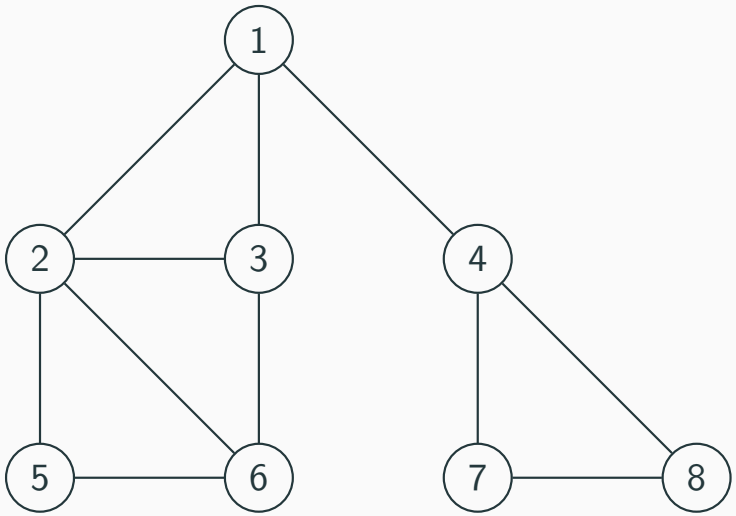
- `rp(2)`. Llamada recursiva, visita 3.
  - `rp(3)`. Llamada recursiva, visita 6.
    - `rp(6)`. Llamada recursiva, visita 5.
      - `rp(5)`. Sin vértices adyacentes.
- `rp(4)`. Llamada recursiva de `rp(1)`, visita 7.
  - `rp(7)`. Llamada recursiva, visita 8.
    - `rp(8)`. Sin vértices que visitar.

Recorrido en anchura

```
procedure RecorridoAnchura(G = (V, A))
    for all v ∈ V do
        marca[v] ← no visitado
    end for
    for all v ∈ V do
        if marca[v] = no visitado then
            ra(v)
        end if
    end for
end procedure
```

```
procedure ra(vértice v)
    Q ← cola vacía
    marca[v] ← visitado
    Insertar v en Q
    while Q ≠ ∅ do
        u ← Primero(Q) y quitarlo
        for all w adyacente a u do
            if marca[w] = no visitado then
                marca[w] ← visitado
            end if
            Insertar w en Q
        end for
    end while
end procedure
```

Recorrido en anchura



Paso	Visitado	Q
1	1	2,3,4
2	2	3,4,5,6
3	3	4,5,6
4	4	5,6,7,8
5	5	6,7,8
6	6	7,8
7	7	8
8	8	

Introducción

Fuerza bruta (búsqueda exhaustiva)

- Explora todas las posibles combinaciones de elementos.
- Sea m_i el tamaño del conjunto S_i, hay $m = m_1 \cdot m_2 \cdot \ldots \cdot m_n$ n-tuplas posibles candidatos a satisfacer la función P.
- El enfoque de la fuerza bruta trataría de formar todas esas tuplas y evaluar cada una de ellas con P, eligiendo la que diera un mejor valor de P o satisfaga P.

Introducción

- Las técnicas de vuelta atrás (*Backtracking*) y ramificación y poda (*Branch & Bound*) exploran los posibles estados de un problema hasta llegar a la solución.
- Pueden implementar un recorrido en profundidad, en anchura o variantes que ayuden en la búsqueda.
- La idea es construir la tupla eligiendo una componente cada vez y, mediante funciones de criterio modificadas $P_i(x_1, \ldots, x_n)$, comprobar si la tupla que se está creando puede llegar a ser una solución.
- La ventaja es que, si a partir de la tupla parcial $(x_1, x_2, \ldots, x_i)$ se deduce que no se podrá construir una solución, entonces se pueden ignorar por completo $m_{i+1} \cdot \ldots \cdot m_n$ posibles tuplas.

Notación

Notación

- **Solución parcial**. Tupla solución para la que aún no se han asignado valores a todos sus componentes.
- **Función de poda**. Aquella que permite identificar cuando una solución parcial conduce (o no) a una solución del problema.
- **Restricciones explícitas**. Reglas que restringen a cada componente x_i de la tupla solución a tomar valores solo en un conjunto dado. Todas las tuplas que satisfacen las restricciones explícitas definen un espacio de soluciones del caso que se está resolviendo.
- **Restricciones implícitas**. Aquellas que determinan cuál (o cuáles) de las tuplas del espacio de soluciones satisface la función de criterio. Es decir, describen la forma en la que las x_i deben relacionarse entre sí. Por tanto, son las restricciones que impone el camino que define la solución.

Notación

- **Árbol de estados**. Organización en árbol del espacio de soluciones.
- **Estado del problema**. Cada uno de los vértices del árbol de estados.
- **Estados solución**. Aquellos estados del problema s para los que el camino desde la raíz a s define una n-tupla en el espacio de soluciones.
- **Estados respuesta**. Aquellos estados solución s para los que el camino desde la raíz hasta s define una tupla que es miembro del conjunto de soluciones (satisfacen las restricciones implícitas) del problema.

Notación

- **Vértice vivo**. Vértice (estado del problema) que ha sido generado, pero del que aún no se han generado todos sus vértices hijos.
- **Vértice muerto (explorado)**. Aquel que ha sido generado, y o bien se ha podado o bien se han generado todos sus hijos.
- **e-vértice (vértice en expansión)**. Vértice vivo del que actualmente se están generando sus hijos.

El problema de la suma de subconjuntos

El problema de la suma de subconjuntos

Enunciado

- Dados $n+1$ números enteros positivos, w_i, $1 \leq i \leq n$, y uno más M, se trata de encontrar todos los subconjuntos de números w_i cuya suma sea igual a M.

Ejemplo

- Sea $n = 4$, con $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$, y $M = 31$.
- Los subconjuntos buscados son:

$$(11, 13, 7) \qquad (24, 7)$$

Pueden existir distintas representaciones de forma que todas las soluciones sean tuplas satisfaciendo ciertas restricciones.

El problema de la suma de subconjuntos

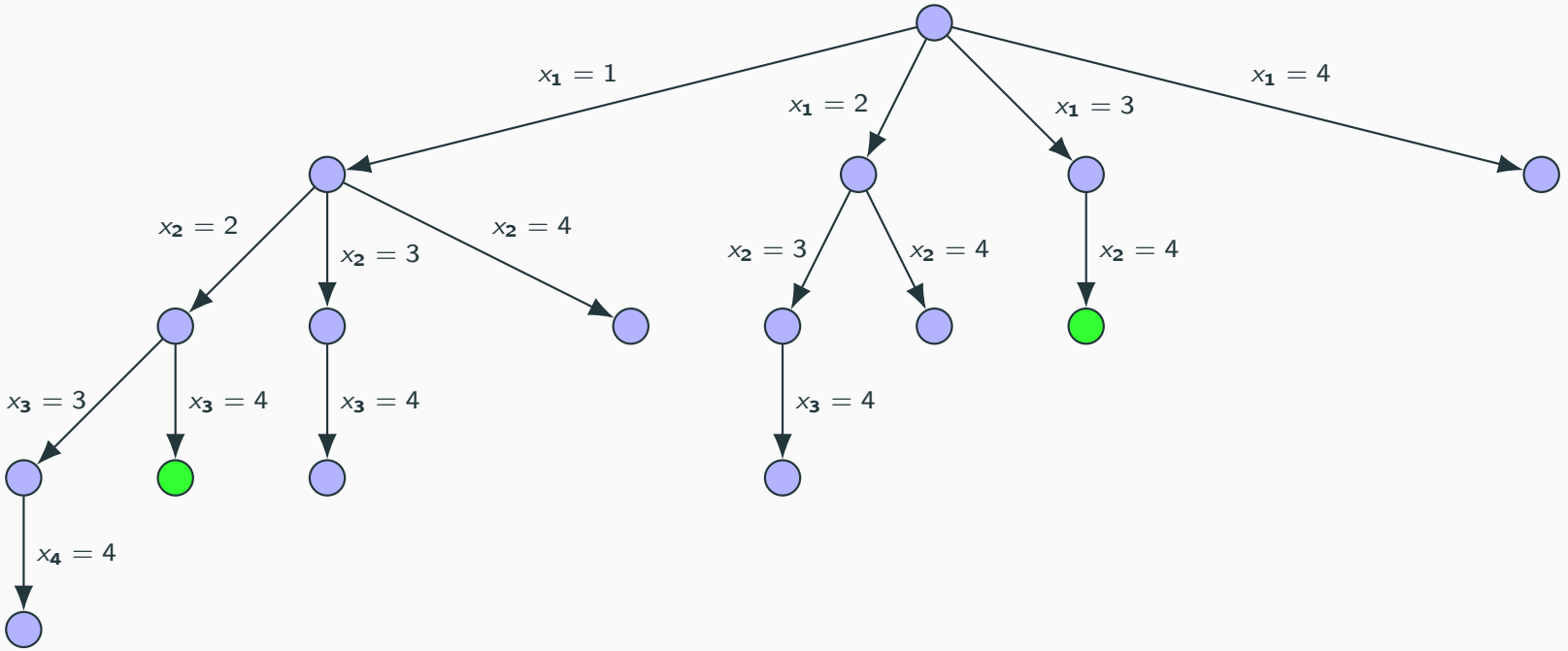
Representación 1

- Una solución se puede representar mediante una tupla que contenga los índices correspondientes w_i.
- Todas las soluciones son k-tuplas $(x_1, x_2, \ldots, x_k)$, $1 \leq k \leq n$, y soluciones diferentes pueden tener tamaños de tupla diferentes.
- Restricciones explícitas: $x_i \in \{j : \ j \text{ es entero y } 1 \leq j \leq n\}$.
- Restricciones implícitas:
  - No puede haber dos componentes con igual valor: $x_i \neq x_j$, para $i \neq j$.
  - La suma de los correspondientes w_i debe ser M.
  - $x_i < x_{i+1}$, para $1 \leq i < n$. Por ejemplo, $(1, 2, 4)$ y $(1, 4, 2)$ representa el mismo subconjunto.

El problema de la suma de subconjuntos

Árbol de estados 1

- Los arcos se etiquetan de modo que uno desde el nivel de vértices i hasta el $i+1$ representa un valor para x_i.
- En esta representación, todos los vértices son estados solución.
- Los posibles caminos son (), (1), (12), (123), (1234), (124), (13), (134), (14), (2), (23), (234), (24), (3), (34), (4).



El problema de la suma de subconjuntos

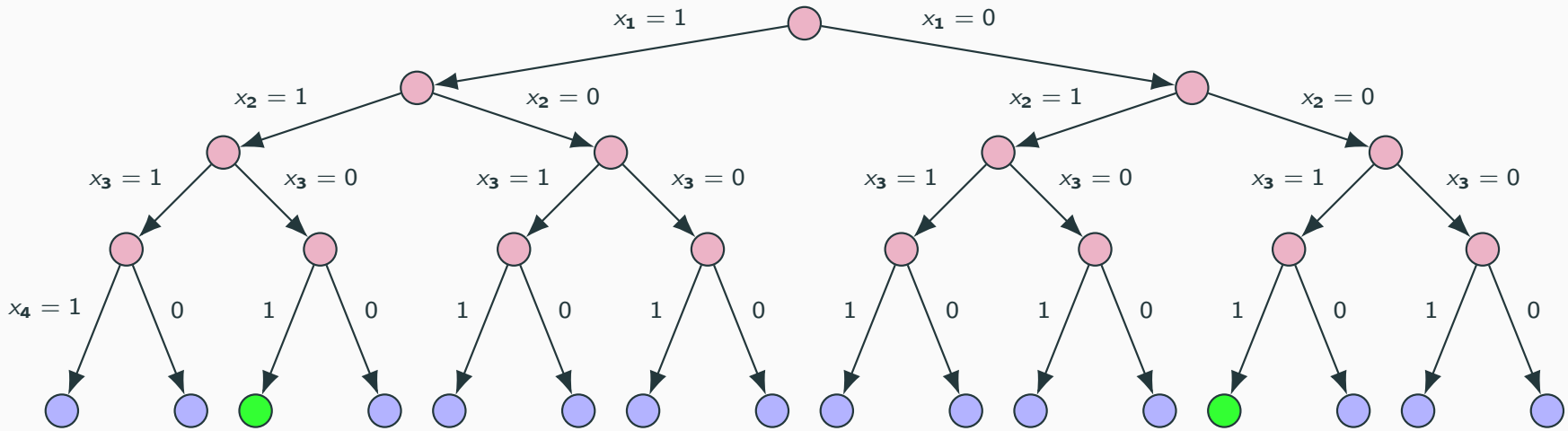
Representación 2

- Cada solución se representa por una n-tupla $(x_1, \ldots, x_n)$ tal que $x_i \in \{0, 1\}$, $1 \leq i \leq n$, y:
  - Si $x_i = 1$, entonces se elige w_i.
  - Si $x_i = 0$, entonces no se elige w_i.
- Las soluciones del anterior caso son $(1, 1, 0, 1)$ y $(0, 0, 1, 1)$.
- Esta formulación expresa todas las soluciones usando un tamaño de tupla fijo.
- Para ambas representaciones, el espacio solución consiste en 2^4 tuplas distintas.

El problema de la suma de subconjuntos

Árbol de estados 2

- Un arco del nivel i al $i+1$ se etiqueta con el valor de x_i (0 o 1).
- Todos los caminos desde la raíz a las hojas definen el espacio solución.
- En esta representación, solo los vértices hoja son estados solución.



El problema de la suma de subconjuntos

Resolución por fuerza bruta

- Implícitamente, se impone una estructura de árbol sobre el conjunto de posibles soluciones (espacio de soluciones).
- La generación de soluciones equivale a efectuar un recorrido del árbol, cuyos vértices (hojas) son posibles soluciones del problema.
- ¿Se puede mejorar el proceso? Sí, si se elimina la necesidad de llegar a todos los vértices del árbol.
- ¿Cuándo? Cuando desde un vértice no se puede alcanzar una solución. En tal caso, se puede podar la rama completa del árbol.
- ¿Cómo?
  1. Vuelta atrás (*Backtracking*).
  2. Ramificación y poda (*Branch & Bound*).

La técnica de vuelta atrás (*Backtracking*)

La técnica de vuelta atrás (*Backtracking*)

- Corresponde a una generación en **profundidad** de estados del problema.
- A partir de un vértice raíz, genera los diferentes estados del problema.
- Al ir generando los estados, se mantiene una lista de vértices vivos.
- Tan pronto como un nuevo hijo h del e-vértice en curso c ha sido generado, este hijo se convierte en un nuevo e-vértice.
- El vértice c se convertirá de nuevo en e-vértice cuando el sub-árbol h haya sido explorado por completo.
- Se emplean funciones de acotación para «matar» los vértices vivos sin tener que generar todos sus vértices hijos.

La técnica de vuelta atrás (*Backtracking*)

- Si un vértice solución s es respuesta, hay tres opciones:
  1. El algoritmo termina, retornando s.
  2. El algoritmo añade s al conjunto de soluciones y continúa el proceso de búsqueda.
  3. El algoritmo evalúa la función objetivo en tal vértice, $P(s)$, actualiza la solución óptima s_{opt} y continúa el proceso de búsqueda.
- Si un vértice solución no es respuesta, el algoritmo continúa el proceso de búsqueda.
- Ello implica deshacer decisiones previas (volver a una llamada recursiva anterior) y tomar en su lugar otras decisiones que conducen a otras ramas del árbol y finalmente a nuevos vértices solución, que serán evaluados.

Implementación (1/2)

- Supongamos que hay que encontrar no solo uno, sino todos los vértices respuesta.
- Sea $(x_1, x_2, \ldots, x_i)$ un camino desde la raíz hasta un vértice del árbol de estados.
- Sea $T(x_1, x_2, \ldots, x_i)$ el conjunto de todos los posibles valores x_{i+1} tales que $(x_1, x_2, \ldots, x_{i+1})$ también es un camino hacia un estado del problema.

Implementación (2/2)

- Existen funciones de acotación B_{i+1} (expresadas como predicados) tales que $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ es falsa para un camino $(x_1, x_2, \ldots, x_{i+1})$ desde el vértice raíz hasta un estado del problema si el camino no se puede extender para llegar a un vértice respuesta.
- $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ indica si $(x_1, x_2, \ldots, x_{i+1})$ satisface las restricciones implícitas del problema.
- Los candidatos para la posición $i + 1$ de la tupla solución $(x_1, x_2, \ldots, x_n)$ son aquellos valores generados por $T(x_1, x_2, \ldots, x_i)$ que satisfacen B_{i+1}.

La técnica de vuelta atrás (*Backtracking*)

Pseudocódigo

```
procedure Backtracking(X[1..n])
    k = 1
    while k > 0 do
        if queda algún X[k] no probado tal que X[k] ∈ T(X[1..k-1])
        y B_k(X[1..k]) = true then
            if X[1..k] es un camino hacia un vértice respuesta then
                Imprimir(X[1..k])
            end if
            k = k + 1
        else
            k = k − 1
        end if
    end while
end procedure
```

La técnica de vuelta atrás (*Backtracking*)

Pseudocódigo (recursivo)

```
procedure RBacktracking(X[1..n], k)
    for all X[k] ∈ T(X[1..k-1]) do
        if B_k(X[1..k]) = true then
            if X[1..k] es un camino hacia un vértice respuesta then
                Imprimir(X[1..k])
            end if
            RBacktracking(X, k+1)
        end if
    end for
end procedure
```

Eficiencia (1/2)

- Depende de cuatro factores:
  - El tiempo necesario para generar el siguiente x_k.
  - El número de valores x_k que satisfagan las restricciones explícitas.
  - El tiempo requerido por las funciones de acotación B_k.
  - El número de x_k que satisfagan las funciones B_k para todo k.
- Las funciones de acotación se consideran buenas si reducen considerablemente el número de vértices que hay que explorar.
- Las buenas funciones de acotación consumen mucho tiempo en su evaluación, por lo que hay que buscar un equilibrio entre el tiempo de evaluación de las funciones de acotación y la reducción del número de vértices generados.

Eficiencia (2/2)

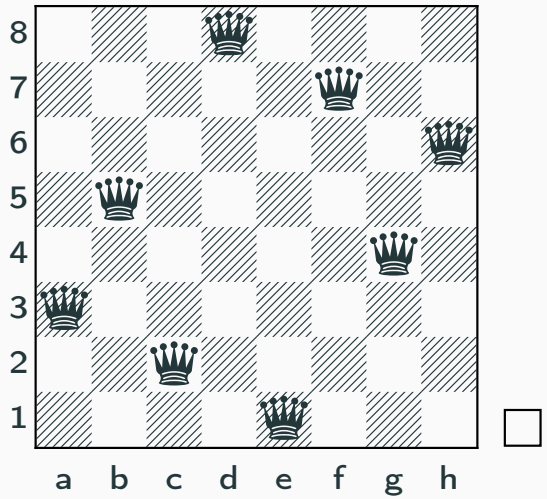
- De los cuatro factores, solo el cuarto varía de un caso a otro (el número de vértices generados).
- En un caso, un algoritmo *backtraking* podría generar solo $O(n)$ vértices, mientras que en otro (parecido) podría generar casi todos los vértices del árbol de estados.
- Si el número de vértices en el espacio solución es 2^n o $n!$, el tiempo del peor caso para el algoritmo será generalmente $O(p(n) \cdot 2^n)$ u $O(q(n) \cdot n!)$ respectivamente, con p y q polinomios en n.
- La importancia del *backtraking* reside en su capacidad para resolver casos con grandes valores de n en muy poco tiempo.
- La dificultad está en predecir la conducta del algoritmo *backtraking* en el caso que se desee resolver.

El problema de las 8 reinas

El problema de las 8 reinas

Enunciado

- Considérese un tablero de ajedrez de 8×8 casillas. El objetivo es colocar 8 reinas en el tablero sin que se amenacen entre sí. Una reina amenaza a las casillas de la misma fila, de la misma columna y de las mismas diagonales.



El problema de las 8 reinas

Representaciones

1. Una 8-tupla de las posiciones de las reinas, siendo cada posición un par (fila, columna).
   - Habría que examinar $\binom{64}{8} = 4\,426\,165\,368$ situaciones.

2. Como cada reina debe estar en una fila diferente, suponemos que la reina i se coloca en la fila i. Las soluciones se representan como una 8-tupla donde x_i es la columna en la que se coloca la reina i.
   - Esto reduce el número de situaciones a examinar a $8^8 = 16\,777\,216$.

El problema de las 8 reinas

Vuelta atrás: componentes de diseño

- **Representación**: $(x_1, x_2, \ldots, x_n)$ es una tupla donde cada componente i representa una fila del tablero y cada valor x_i es la columna donde se colocará la reina de la i-ésima fila.
- **Restricciones explícitas**: $x_i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq n$.
- **Restricciones implícitas**:
  - No puede haber dos reinas en la misma columna: $x_i \neq x_j$, $i \neq j$. Esto reduce el tamaño a $8! = 40\,320$.
  - No puede haber dos reinas en la misma diagonal: $|x_j - x_i| \neq |j - i|$, $i \neq j$.

El problema de las 8 reinas

Algoritmo

```
1  bool factible(int *reinas, int n, int k){
2    bool valido = true;
3    int i = 0;
4
5    while (i < k && valido){
6      if ((reinas[i] == reinas[k]) ||
7        (abs(k-i) == abs(reinas[k] - reinas[i]))){
8        valido = false;
9      }
10     i++;
11   }
12
13   return valido;
14 }
```

El problema de las 8 reinas

Algoritmo

```
1  void reinas8(int *reinas, int n, int k) {
2    if (k == n){
3      imprimir_solucion(reinas, n);
4    }
5    else{
6      for (reinas[k] = 0; reinas[k] < n; reinas[k]++){
7        if (factible(reinas, n, k)){
8    reinas8(reinas, n, k+1);
9        }
10     }
11   }
12 }
```

El problema del coloreo de un grafo

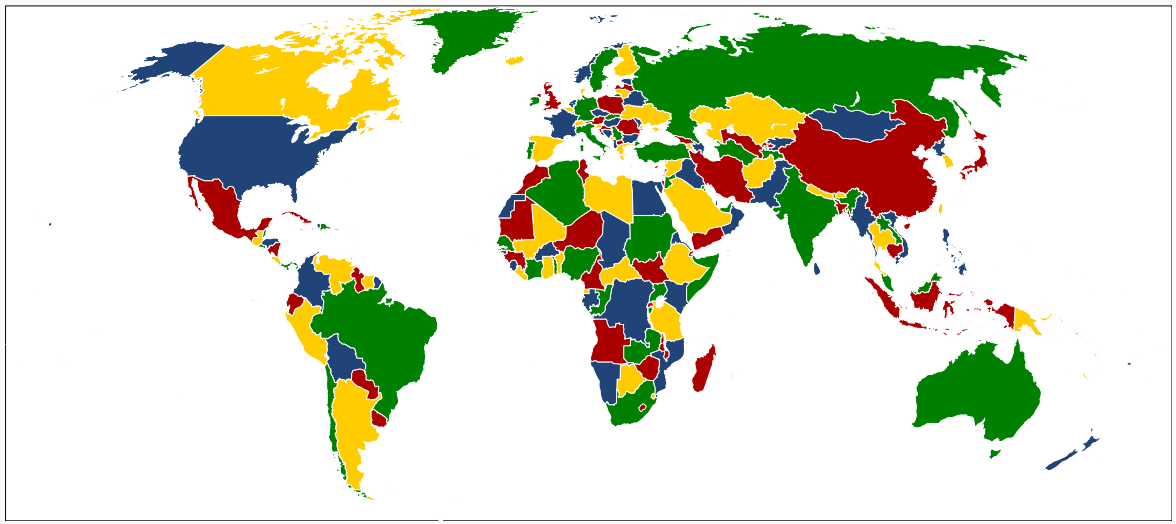
El problema del coloreo de un grafo

Enunciado

- Sea $G = (V, A)$ un grafo no dirigido y m un número entero positivo. Queremos saber si los vértices del grafo se pueden colorear de forma que:
  - No haya dos vértices adyacentes con el mismo color.
  - Solo se utilicen n colores para esa tarea.
- Este es el problema de la m-colorabilidad.
- El problema de optimización de la m-colorabilidad pregunta por el menor número m con el que el grafo se puede colorear.
- Al número entero m se le denomina número cromático del grafo.

El problema del coloreo de un grafo

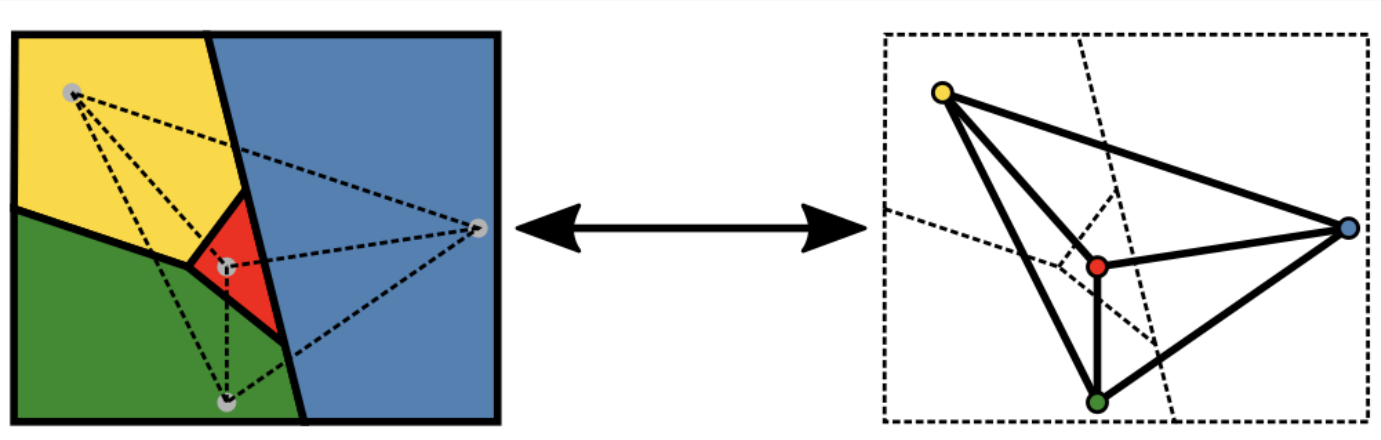
- Un grafo es plano si y solo si puede pintarse en un plano de modo que ningún par de aristas se corten entre sí.
- Un caso especial y famoso del problema de la m-colorabilidad es el problema de los cuatro colores para grafos planos que, dado un mapa cualquiera, consiste en saber si ese mapa podrá pintarse con tan solo cuatro colores y de forma que no haya dos zonas colindantes con el mismo color (sí se puede).



El problema del coloreo de un grafo

El coloreo de un mapa

- Queremos colorear un mapa con cuatro colores a lo sumo.
- Los países adyacentes deben tener colores diferentes.
- Este problema es fácilmente traducible a la nomenclatura de grafos.



El problema del coloreo de un grafo

Vuelta atrás: componentes de diseño

- **Representación**: $(x_1, x_2, \ldots, x_n)$ es una tupla donde cada componente x_i representa el color del vértice i, siendo n el número de vértices.
- **Restricciones explícitas**: $x_i \in \{1, 2, \ldots, m\}$.
- **Restricciones implícitas**: $x_i \neq x_j$ si i y j son vértices (países) adyacentes.
- **Árbol de estados**: el espacio de estados es un árbol de grado m y altura $n + 1$ en el que cada vértice del nivel i tiene m hijos correspondientes a las m posibles asignaciones para x_i y en el que los vértices del nivel $n + 1$ son vértices hoja.

El problema del coloreo de un grafo

Algoritmo

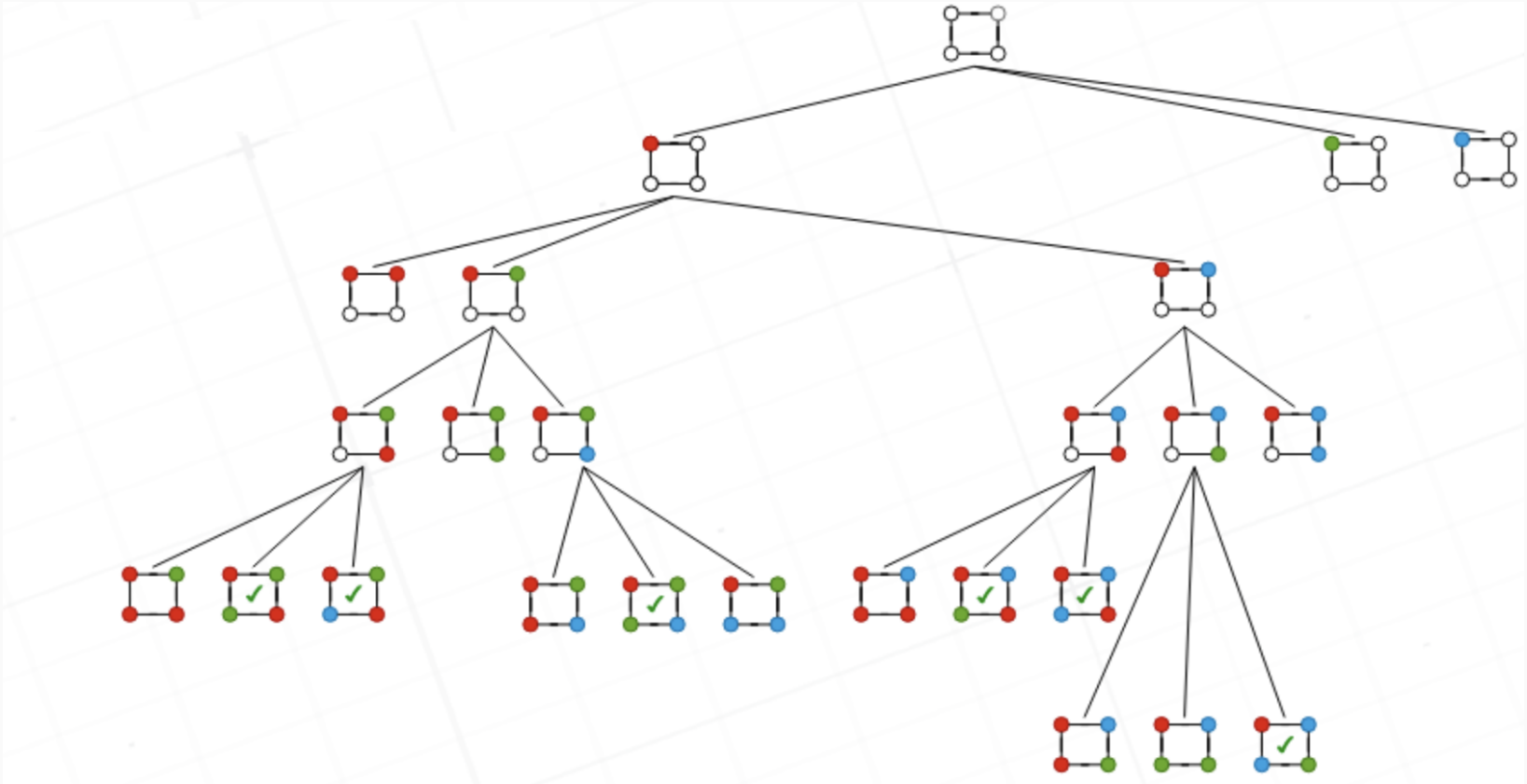
```
1  bool factible(int v, bool grafo[V][V], int color[], int c) {
2    int i = 0;
3    bool es_factible = true;
4    while (i < V && es_factible){
5      if (grafo[v][i] && c == color[i]){
6        es_factible = false;
7      }
8      i++;
9    }
10   return es_factible;
11 }
```

El problema del coloreo de un grafo

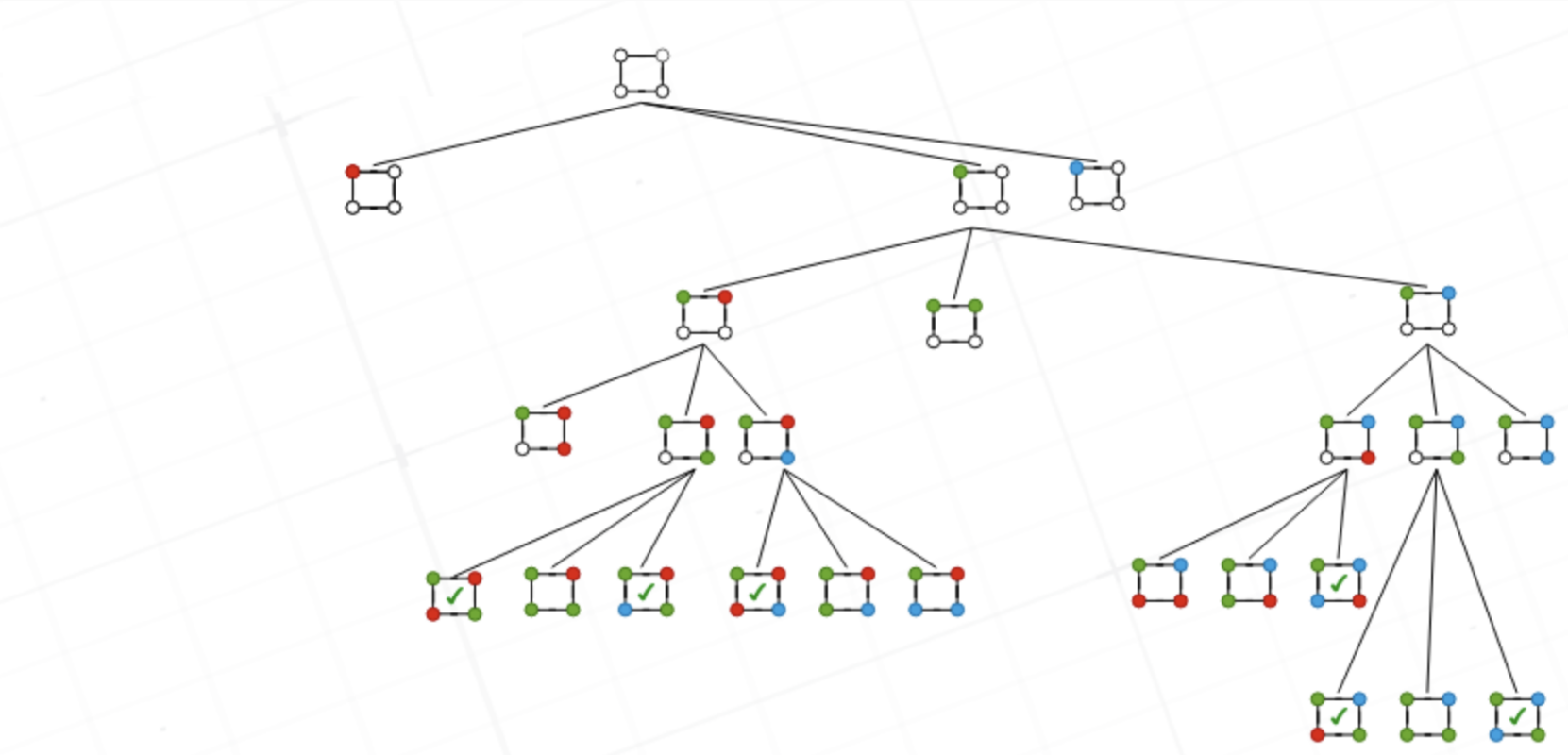
Algoritmo

```
1  void coloreo(bool grafo[][V], int m, int *color, int k){
2    if (k == V){
3      imprimir_sol(color);
4    }
5    else{
6      for (int c = 1; c <= m; c++){
7        if (factible(grafo, c, color, k)){
8    color[k] = c;
9    coloreo(grafo, m, color, k+1);
10   color[k] = 0;
11       }
12     }
13   }
14 }
```

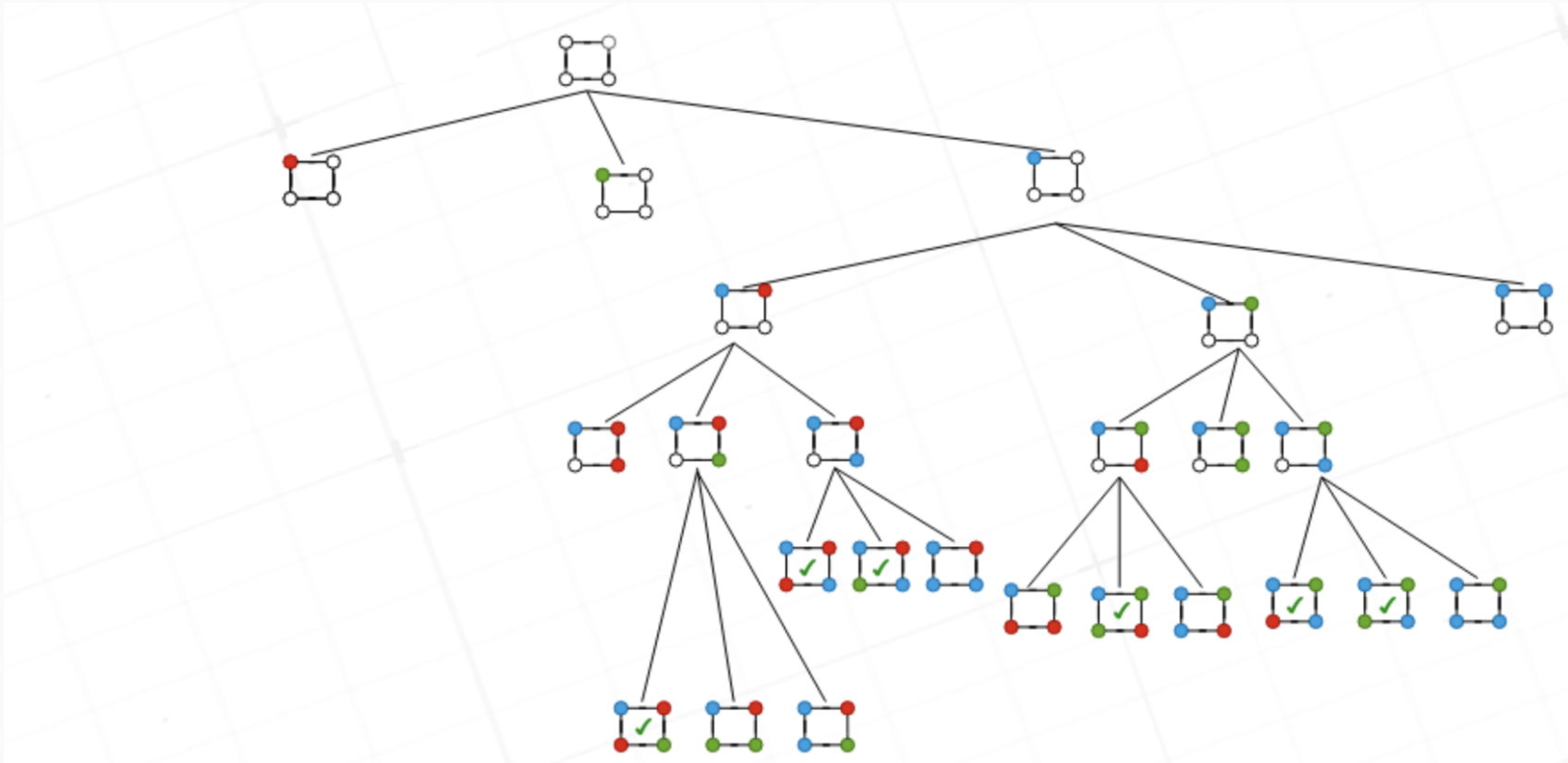
El problema del coloreo de un grafo



El problema del coloreo de un grafo



El problema del coloreo de un grafo



El problema del viajante de comercio

El problema del viajante de comercio

Enunciado

- Se conocen las distancias entre un determinado número de ciudades.
- Partiendo de una de ellas, un viajante debe visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible.

Enunciado formal

- Dado un grafo $G = (V, A)$ no dirigido, conexo, ponderado y completo, y dado uno de sus vértices v_0, encontrar el ciclo hamiltoniano de coste mínimo que comienza y termina en v_0.

El problema del viajante de comercio

Vuelta atrás: componentes de diseño

- **Representación**: $(x_1, x_2, \ldots, x_n)$ es una tupla donde cada componente x_i se asocia al vértice que hay que visitar en la i-ésima posición en el ciclo propuesto, siendo n el número de vértices.
- **Restricciones explícitas**:
  - $x_1 = 1$ (para evitar imprimir el mismo ciclo n veces).
  - $x_i \in \{2, 3, \ldots, n\}$, $i = 2, 3, \ldots, n$.
- **Restricciones implícitas**: $x_i \neq x_j$, para $i \neq j$ (no puede visitarse el mismo vértice dos veces).

El problema del viajante de comercio

Algoritmo

```
1  void viajante(int g[][V], int *x, int act, int k,
2     int coste, int *ciclo, int& min_coste){
3    if (k == V && g[act][0]){
4      if (coste + g[act][0] < min_coste){
5        min_coste = coste + g[act][0];
6        for(int i = 1; i < V; i++)
7    ciclo[i] = x[i]+1;
8      }
9    }
10   else{
11     for (int i = 1; i < V; i++){
12       if (x[i] == -1 && g[act][i]){
13   x[i] = k;
14   viajante(g,x,i,k+1,coste+g[act][i],ciclo,min_coste);
15   x[i] = -1;
16       }
17     }
18   }
19 }
```

La técnica de ramificación y poda (*Branch & Bound*)

Branch & Bound (B&B) (1/2)

- Es una técnica muy similar a la de *backtracking*.
- 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 y para jugar a juegos.
- Los algoritmos generados por esta técnica son normalmente de orden exponencial, o peor, en su peor caso.

Branch & Bound (B&B) (2/2)

- Su aplicación en casos muy grandes ha demostrado ser eficiente (incluso más que *backtracking*).
- Puede ser vista como una generalización (o mejora) de la técnica de *backtracking*.
- La principal novedad es que habrá una estrategia de ramificación.
- Se tratará como un aspecto importante las técnicas de poda, para eliminar vértices que no lleven a soluciones óptimas.
- La poda se realiza estimando en cada vértice cotas del beneficio (coste) que podemos obtener a partir de este.

La técnica de ramificación y poda (*Branch & Bound*)

Diferencia con *backtracking*

- En *backtracking*, tan pronto como se genera un nuevo hijo del e-vértice actual, este hijo pasa a ser el nuevo e-vértice.
- En B&B, se generan todos los hijos del e-vértice antes de que cualquier otro vértice vivo pase a ser el nuevo e-vértice (no se realiza una búsqueda en profundidad).

En consecuencia

- En *backtracking*, los únicos vértices vivos son los que están en el camino de la raíz al vértice en curso.
- En B&B, puede haber más vértices vivos (se guardan en una estructura de datos auxiliar: **lista de vértices vivos**).

La técnica de ramificación y poda (*Branch & Bound*)

Más diferencias con *backtracking*

- En *backtracking*, el test de comprobación realizado por las funciones de poda nos indica únicamente si un nodo concreto nos puede llevar a una solución o no.
- En B&B, se acota el valor de la solución a la que nos puede conducir un vértice concreto.
- Esta acotación permite:
  - Podar el árbol (si sabemos que no nos va a conducir a una solución mejor que la que tenemos).
  - Establecer el orden de ramificación, de modo que comenzaremos explorando las ramas más prometedoras del árbol.

La técnica de ramificación y poda (*Branch & Bound*)

Descripción general (1/2)

- B&B es un método de búsqueda general que se aplica de la siguiente forma:
  - Explora un árbol comenzando a partir de un problema raíz (el problema original con su espacio de soluciones completo).
  - Entonces, aplica procedimientos de costa inferiores y superiores al problema raíz.
  - Si las costas cumplen las condiciones que se hayan establecido, habremos encontrado la solución óptima y el procedimiento termina.
  - Si no, entonces el espacio de soluciones se divide en dos o más, dando lugar a distintos subproblemas.
  - Estos subproblemas particionan el espacio de soluciones. La búsqueda se desarrolla en cada uno de ellos.
  - El método se aplica recursivamente a los subproblemas.

Descripción general (2/2)

- Si se encuentra un solución óptima para un subproblema, esta será una solución factible para el problema completo, pero no necesariamente el óptimo global.
- Cuando en un vértice (subproblema) la cota superior es menor que el mejor valor conocido en la región, no puede existir un óptimo global en el sub-espacio de la región factible asociada a ese vértice.
- Por tanto, ese vértice puede ser eliminado (podado) en posteriores consideraciones.
- La búsqueda sigue hasta que se examinan o podan todos los vértices, o hasta que se alcanza algún criterio preestablecido sobre el mejor valor encontrado y las cotas superiores de los subproblemas no resueltos aún.

La técnica de ramificación y poda (*Branch & Bound*)

Para cada vértice i se tiene:

- Una **cota superior**, $CS(i)$, y una **cota inferior**, $CI(i)$, del beneficio (o coste) óptimo que podemos alcanzar a partir de ese vértice.
  - Determinan cuándo se puede realizar una poda.
- Una **estimación del beneficio (o coste)** óptimo que se puede encontrar a partir de ese vértice. Puede ser una media de las anteriores.
  - Ayuda a decidir qué parte del árbol evaluar primero.
- Una **estrategia de poda**:
  - Suponemos un problema de maximización.
  - Hemos recorrido varios vértices $1..n$, estimando para cada uno la cota superior $CS(j)$ y la cota inferior $CI(j)$, respectivamente, para $j \in \{1, \ldots, n\}$.
  - Hay dos casos.

La técnica de ramificación y poda (*Branch & Bound*)

Caso 1

- Si a partir de un vértice i se puede obtener una solución válida, entonces **podar tal vértice si**:

$$CS(i) \leq CI(j) \qquad \text{para algún vértice } j \text{ generado en el árbol.}$$

- Ejemplo: problema de la mochila usando un árbol binario:
  - A partir de i, podemos encontrar un beneficio máximo de $CS(i) = 4$.
  - A partir de j, se garantiza un beneficio mínimo de $CI(j) = 5$.
  - Podemos podar i sin perder ninguna solución óptima.

Caso 2

- Si se obtiene una posible solución válida para el problema con un beneficio $B(j)$, entonces podemos **podar un vértice i si**:

$$CS(i) \leq B(j) \qquad \text{para algún vértice } j\text{, solución final (factible).}$$

- Ejemplo: problema de la mochila usando un árbol binario:
  - A partir de i, podemos encontrar un beneficio máximo de $CS(i) = 4$.
  - Se ha encontrado una solución válida cuyo beneficio es 6.
  - Podemos podar i sin perder ninguna solución óptima.

Estrategias de ramificación (1/2)

- Normalmente, el árbol de estados es implícito, no se almacena en ningún lugar.
- Para realizar el recorrido, se usa una **lista de vértices vivos (LVV)**.
- La LVV contiene todos los vértices que han sido generados pero que no han sido explorados todavía (vértices pendientes de tratar por el algoritmo).
- Según como se implemente la LVV, el recorrido será de un tipo u otro.

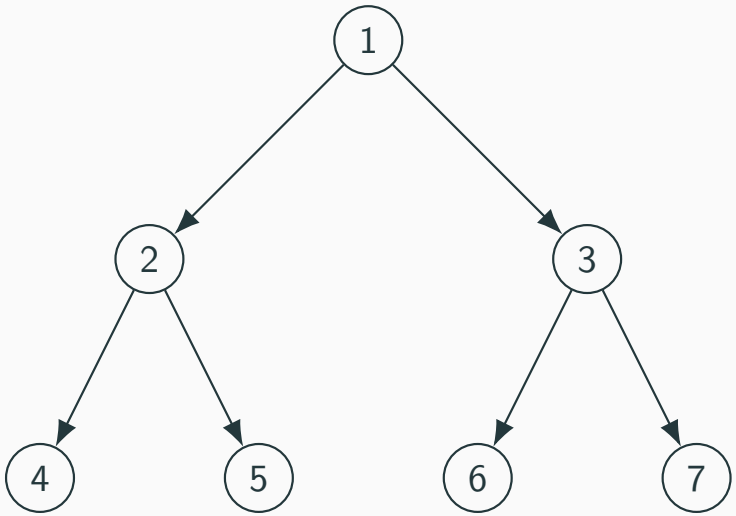
Estrategias de ramificación (2/2)

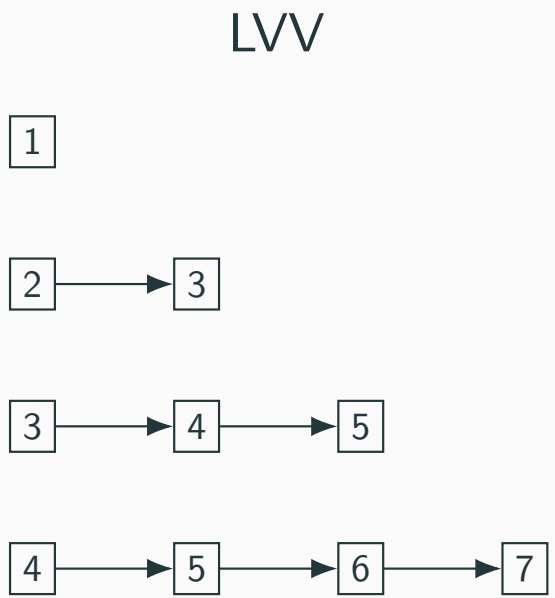
- Estrategias que no tienen en cuenta los beneficios (costes), realizan una búsqueda «a ciegas»:
  - **Estrategia FIFO** (First In, First Out).
  - **Estrategia LIFO** (Last In, First Out).
- Estrategias que usan estimaciones de beneficios (costes), exploran primero los vértices más prometedores:
  - **Estrategias LC** (Least Cost).
  - **Estrategias MB** (Maximum Benefit).

La técnica de ramificación y poda (*Branch & Bound*)

Estrategia de ramificación FIFO

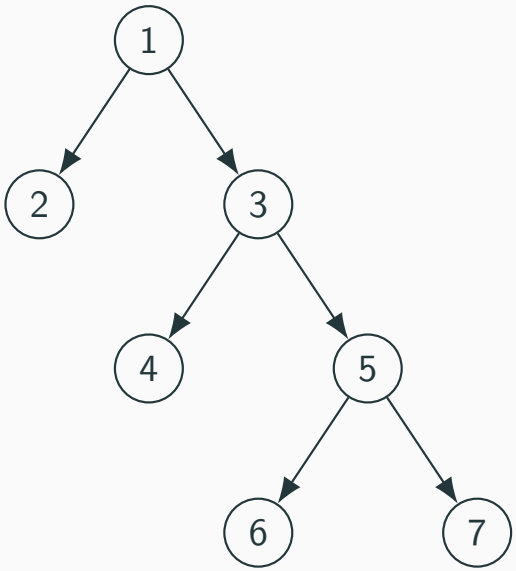
- La lista de vértices vivos es una cola.
- El recorrido del árbol es en anchura.

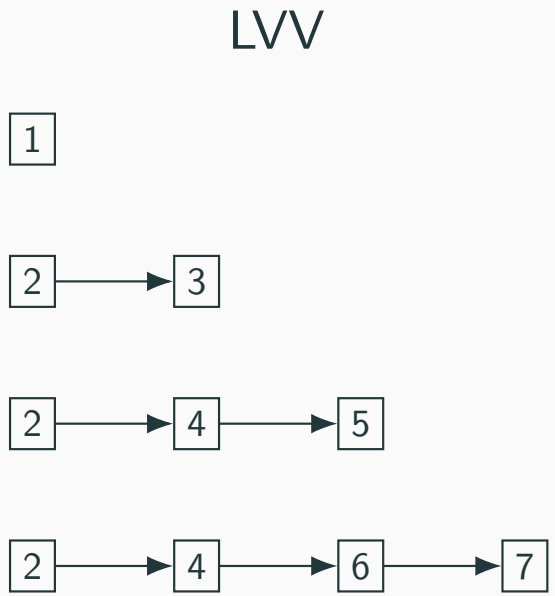




Estrategia de ramificación LIFO

- La lista de vértices vivos es una pila.
- El recorrido del árbol es en profundidad.





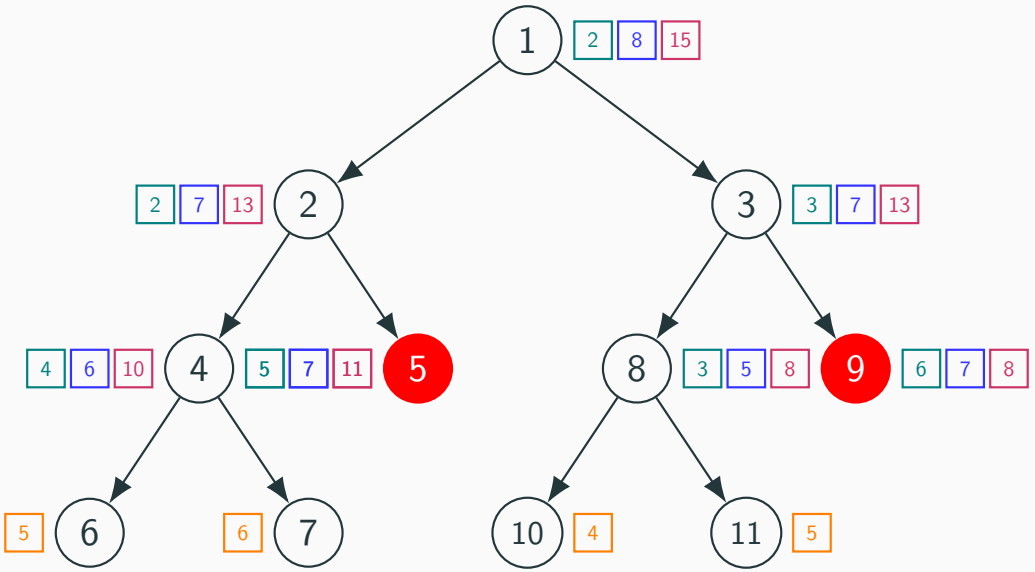
Estrategias de ramificación LC / MB

- En caso de empate (de beneficio o coste estimado), este se deshace usando un criterio FIFO o LIFO:
  - **Estrategia LC-FIFO / MB-FIFO**. Seleccionar el primero que se introdujo en la LVV (de los que empatan).
  - **Estrategia LC-LIFO / MB-LIFO**. Seleccionar el último que se introdujo en la LVV (de los que empatan).
- En cada vértice se puede tener: cota inferior de coste (beneficio), coste (beneficio) estimado, y cota superior del coste (beneficio).
- Podar según los valores de las cotas (CS, CI).
- Ramificar según los valores estimados (E).

La técnica de ramificación y poda (*Branch & Bound*)

Ejemplo: B&B usando LC-FIFO (minimización)

- Para realizar la poda se usa una variable C.
- Esta contiene el valor de la menor de las cotas superiores hasta ese momento o de alguna solución final ya encontrada.
- Si para algún vértice i, $CI(i) \geq C$, entonces podar i.



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

La técnica de ramificación y poda (*Branch & Bound*)

Pseudocódigo (minimización)

```
procedure BranchAndBound(NodoRaiz, Sol)
    LVV ← {NodoRaiz}
    C ← CS(NodoRaiz)
    Sol ← ∅
    while LVV ≠ ∅ do
        X ← Seleccionar(LVV) // Criterio FIFO, LIFO, LC-FIFO, LC-LIFO
        LVV ← LVV − {X}
        if CI(X) < C then // Si no se cumple, se poda X
            for all Y hijo de X do
                if Y es una solución final mejor que Sol then
                    Sol ← Y
                    C ← Min(C,Coste(Y))
                else if Y no es solución final y CI(Y) < C then
                    LVV ← LVV + {Y}
                    C ← Min(C,CS(Y))
                end if
            end for
        end if
    end while
end procedure
```

Observaciones

- Solo se comprueba el criterio de poda cuando se introduce o se saca un vértice de la LVV.
- Si un hijo de un vértice es una solución final, entonces no se introduce en la LVV.
- Se comprueba si esa solución es mejor que la actual y, en tal caso, se actualiza C y se guarda como mejor solución hasta el momento.

Tiempo de ejecución

- Depende de:
  - El número de vértices recorridos. Depende de la efectividad de la poda.
  - El tiempo empleado en cada vértice. Tiempo necesario para hacer las estimaciones de coste y gestionar la LVV en función de la estrategia de ramificación.
- En el peor caso, el tiempo de ejecución de un algoritmo B&B es igual al de un algoritmo de *backtracking* (o peor si tenemos en cuenta el tiempo que requiere la LVV).
- En el caso promedio, no obstante, se suelen obtener mejoras con respecto a *backtracking*.

La técnica de ramificación y poda (*Branch & Bound*)

¿Cómo hacer que un algoritmo B&B sea más eficiente?

- Hacer estimaciones de costo muy precisas:
  - Se realiza una poda exhaustiva del árbol de estados.
  - Se recorren menos vértices.
  - Se emplea mucho tiempo en realizar las estimaciones.
- Hacer estimaciones de coste poco precisas:
  - Se emplea poco tiempo en cada vértice.
  - No se poda mucho.
  - El número de vértices explorados puede ser muy elevado.

Conclusión

- Se debe buscar un equilibrio entre la precisión de las cotas y el tiempo empleado en calcularlas.

El problema de la asignación de tareas

El problema de la asignación de tareas

Enunciado

- Supongamos que disponemos de n trabajadores y n tareas.
- Sea $b_{ij} > 0$ el beneficio de asignar la tarea j al trabajador i.
- Una asignación de tareas puede expresarse como una asignación de los valores 0 o 1 a las variables x_{ij}:
  - $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j.
  - $x_{ij} = 1$ indica que sí.
- Una asignación es válida si a cada trabajador solo le corresponde una tarea y cada tarea está asignada a un único trabajador.
- Dada una asignación válida, se define su beneficio como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} \cdot c_{ij}$$

- Diremos que una asignación es óptima si es de máximo beneficio.

El problema de la asignación de tareas

Ramificación y poda: componentes de diseño

- **Representación**: $(x_1, x_2, \ldots, x_n)$ es una tupla donde cada componente x_i se asocia a la tarea asignada al trabajador i, siendo n el número de trabajadores y de tareas.
- **Restricciones explícitas**: $x_i \in \{1, 2, \ldots, n\}$, $i = 1, 2, \ldots, n$.
- **Restricciones implícitas**: $x_i \neq x_j$, para $i \neq j$ (no puede asignarse la misma tarea a dos trabajadores distintos).

El problema de la asignación de tareas

Cálculo de cotas

- **Cota inferior** CI: beneficio acumulado hasta ese momento.
- **Cota superior** CS: cota inferior más las restantes asignaciones con el máximo global.
- **Estimación del beneficio**: la media de las cotas.

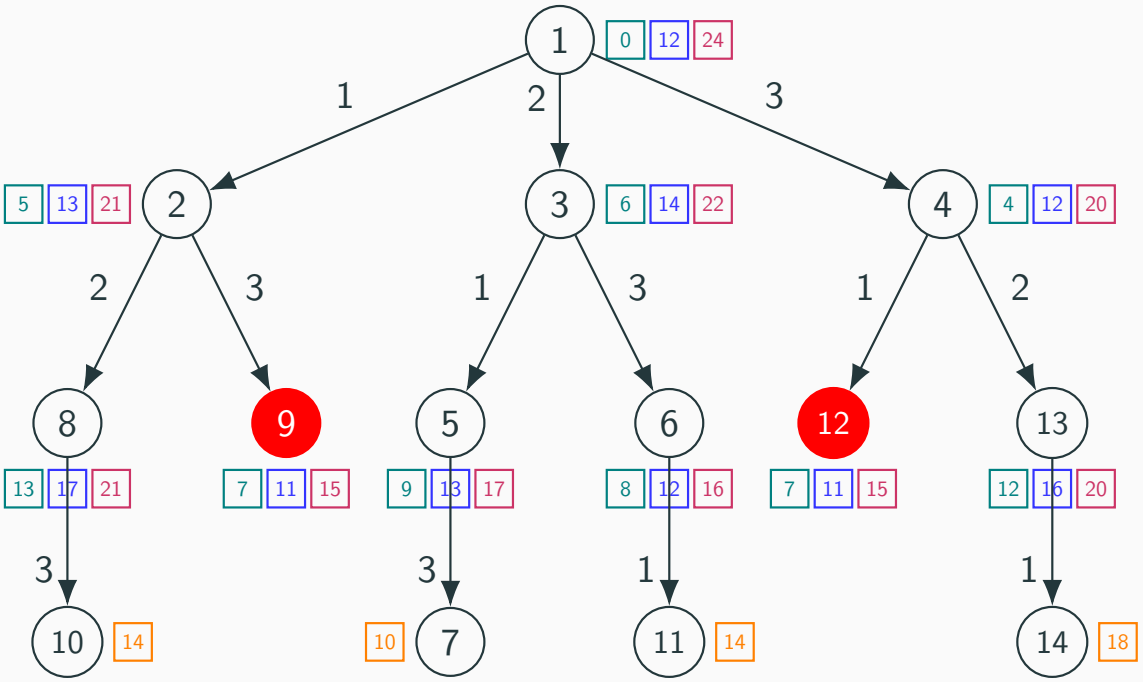
Estrategias de ramificación y poda (maximización)

- **Variable de poda** C: valor de la mayor cota inferior o solución final del problema encontrada hasta ahora.
- **Condición de poda**: podar el vértice i si $CS(i) \leq C$.
- **Estrategia de ramificación**: estrategia MB-LIFO.

El problema de la asignación de tareas

Ejemplo

B	1	2	3
1	5	6	4
2	3	8	2
3	6	5	1



C	LVV
0	1
6	3 → 2 → 5
9	5 → 2 → 6 → 4
10	2 → 6 → 4
13	8 → 6 → 4 → 9
14	6 → 4 → 9
14	4 → 9
14	13 → 9 → 12
18	9 → 12
18	12
18	

El problema de la asignación de tareas

Cálculo de cotas (mejoradas)

- **Cota inferior** CI: beneficio acumulado hasta ese momento más el resultado de asignar a cada persona la tarea libre que proporciona un mayor beneficio.
- **Cota superior** CS: asignar las tareas con mayor beneficio (aunque se repitan).
- **Estimación del beneficio**: la media de las cotas.

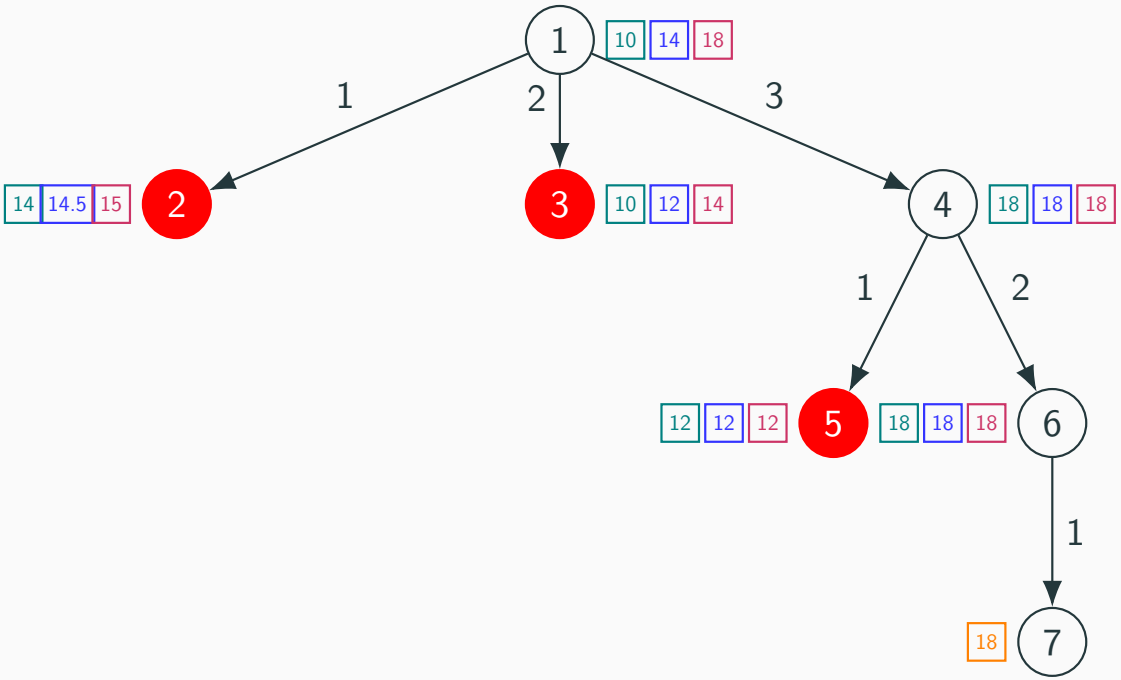
Estrategias de ramificación y poda (maximización)

- **Variable de poda** C: valor de la mayor cota inferior o solución final del problema encontrada hasta ahora.
- **Condición de poda**: podar el vértice i si $CS(i) \leq C$.
- **Estrategia de ramificación**: estrategia MB-LIFO.

El problema de la asignación de tareas

Ejemplo

B	1	2	3
1	5	6	4
2	3	8	2
3	6	5	1



C	LVV
10	1
18	4 → 2 → 3
18	6 → 2 → 5 → 3
18	2 → 5 → 3
18	5 → 3
18	3
18	

El problema del viajante de comercio

El problema del viajante de comercio

Enunciado

- Se conocen las distancias entre un determinado número de ciudades.
- Partiendo de una de ellas, un viajante debe visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible.

Enunciado formal

- Dado un grafo $G = (V, A)$ no dirigido, conexo, ponderado y completo, y dado uno de sus vértices v_0, encontrar el ciclo hamiltoniano de coste mínimo que comienza y termina en v_0.

El problema del viajante de comercio

Ramificación y poda: componentes de diseño

- **Representación**: $(x_1, x_2, \ldots, x_n)$ es una tupla donde cada componente x_i se asocia con el vértice que hay que visitar en la i-ésima posición.
- **Restricciones explícitas**:
  - $x_1 = 1$ (se comienza en el vértice 1).
  - $x_i \in \{2, \ldots, n\}$, $i = 2, \ldots, n$.
- **Restricciones implícitas**: $x_i \neq x_j$, para $i \neq j$ (no puede visitarse el mismo vértice más de una vez).

El problema del viajante de comercio

Árbol de espacio de estados

- 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 dos que ya se han visitado.
- Y así sucesivamente hasta el nivel $n-1$ que incluirá al vértice que no ha sido visitado.

El problema del viajante de comercio

Cálculo de cotas

- **Cota inferior** CI: el mejor coste será el de la arista adyacente que tenga el menor valor. La suma de los costes asociados a las mejores aristas incidentes en cada vértice aún por incluir en el ciclo hamiltoniano más el coste del camino ya recorrido ofrece una estimación válida para tomar decisiones.

Estrategias de ramificación y poda (minimización)

- **Variable de poda** C: valor de la menor cota superior o solución final del problema encontrada hasta ahora.
- **Condición de poda**: podar el vértice i si $CI(i) \geq C$.
- **Estrategia de ramificación**: estrategia LC-LIFO.

El problema del viajante de comercio

Ejemplo

- Dada la siguiente matriz de adyacencias, ¿cuál es el coste mínimo del ciclo hamiltoniano que comienza y termina en el vértice 1?

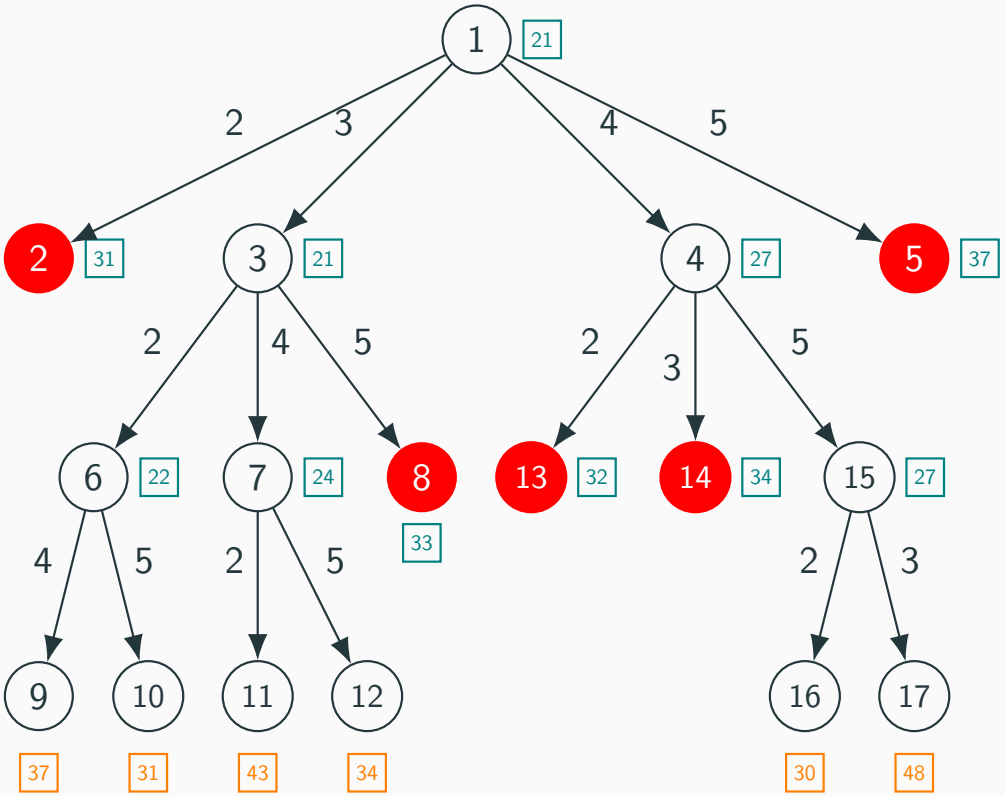
$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$

- Aplicamos el algoritmo voraz para obtener una primera solución del problema.
- La solución calculada por el algoritmo voraz es $1-3-2-5-4-1$, con un coste de 31.
- Por tanto, $C = 31$.

El problema del viajante de comercio

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



C	LVV
31	1
31	3 → 4
31	6 → 7 → 4
31	7 → 4
31	4
31	15
30	

Solución: 1-4-5-2-3-1

Coste: 30

Bibliografía

Bibliografía

Aclaración

- El contenido de las diapositivas es esquemático y representa un apoyo para las clases teóricas.
- Se recomienda completar los contenidos del tema 1 con apuntes propios tomados en clase y con la bibliografía principal de la asignatura.

Por ejemplo


G. Brassard and P. Bratley.
***Fundamentals of Algorithmics*.**
Prentice Hall, Englewwod Cliffs, New Jersey, 1996.


J. L. Verdegay.
***Lecciones de Algorítmica*.**
Editorial Técnica AVICAM, 2017.