BACKTRACKING (VUELTA ATRÁS)

EL ESQUEMA DE BACKTRACKING ES UNA TÉCNICA GENERAL DE EXPLORACIÓN EXHAUSTIVA DE ESPACIOS DE SOLUCIONES

- PARA BUSCAR UNA SOLUCIÓN
- PARA BUSCAR TODAS LAS SOLUCIONES
- PARA BUSCAR LA SOLUCIÓN ÓPTIMA

LAS SOLUCIONES DEBEN PODER EXPRESARSE COMO n-TUPLAS

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

DONDE $x_i \in S_i$, S_i FINITO

BACKTRACKING

LAS CONDICIONES $x_i \in S_i$ SE DENOMINAN RESTRICCIONES EXPLÍCITAS.

SEA $N_i = |S_i|$. ENTONCES EL ESPACIO DE SOLUCIONES CONTIENE

$$N = N_1 \times \dots \times N_n = \prod_{i=1}^{n} N_i$$

ALTERNATIVAS. SI $N_i \geqslant 2$ ENTONCES $N = \Omega(2^n)$.

EL BACKTRACKING PERMITE REDUCIR SUSTANCIALMENTE EL COSTE DE LA BÚSQUEDA PUESTO QUE PUEDE DESESTIMAR "PORCIONES" GRANDES DEL ESPACIO DE SOLUCIONES.

POR EJEMPLO, PUEDE EVITARSE LA EXPLORACIÓN DE UN SUBESPACIO CORRESPONDIENTE A

$$(x_1, x_2, \dots, x_k), \quad k < n$$

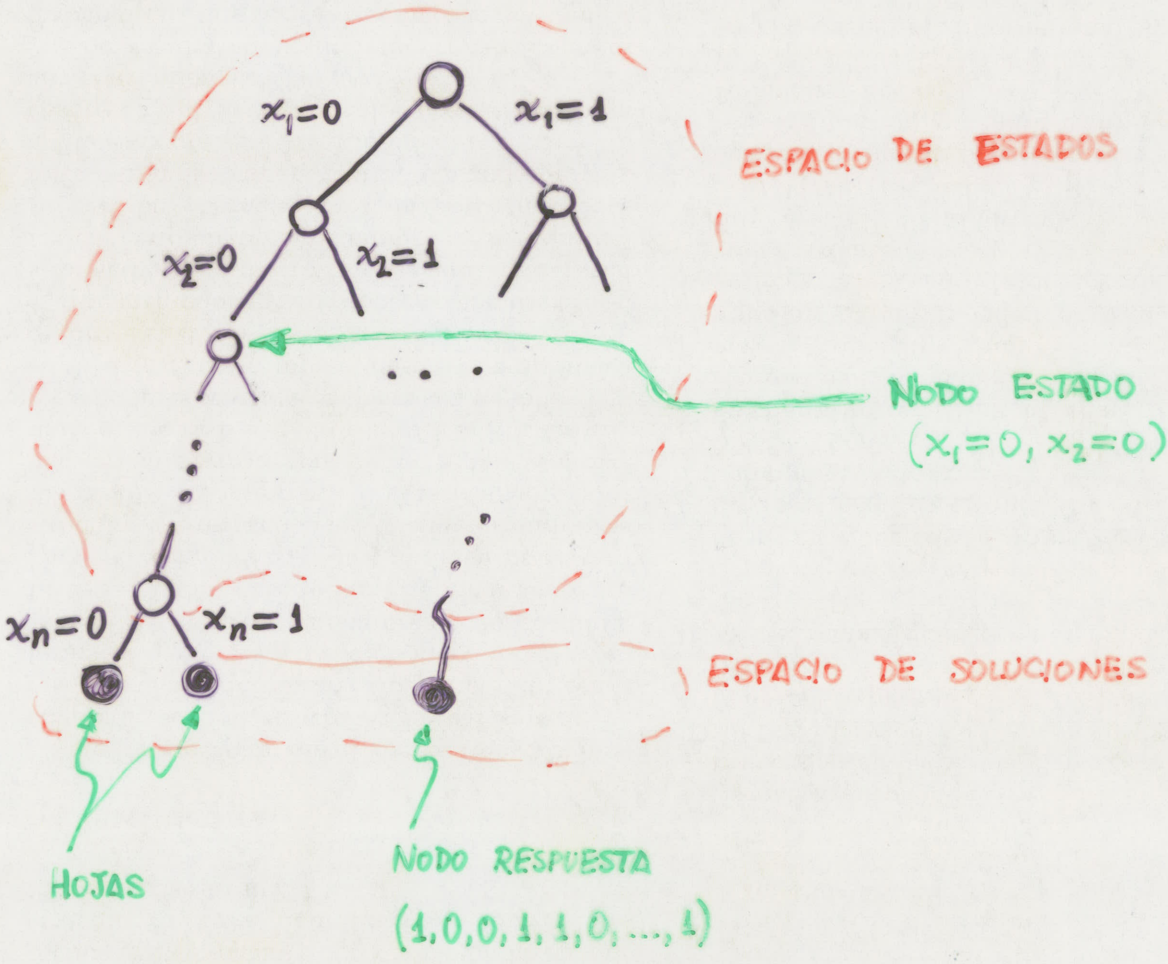
SI SABEMOS QUE NO CONDUCE A NINGUNA SOLUCIÓN.

BACKTRACKING

BACKTRACKING RECORRE $\boxed{\text{EN PROFUNDIDAD}}$ ≡ $\boxed{\text{EN PREORDEN}}$ EL ÁRBOL DE ESTADOS. SE DISPONE DE UN PROCEDIMIENTO completable QUE NOS PERMITE DECIDIR SI UNA SOLUCIÓN PARCIAL

$$(x_1, \dots, x_k) \quad k<n$$

ES POTENCIALMENTE EXTENDIBLE A UNA SOLUCIÓN.



BACKTRACKING

COLOCAR 8 REINAS EN EL TABLERO 8×8 DE MANERA QUE NINGUNA ATAQUE A OTRA

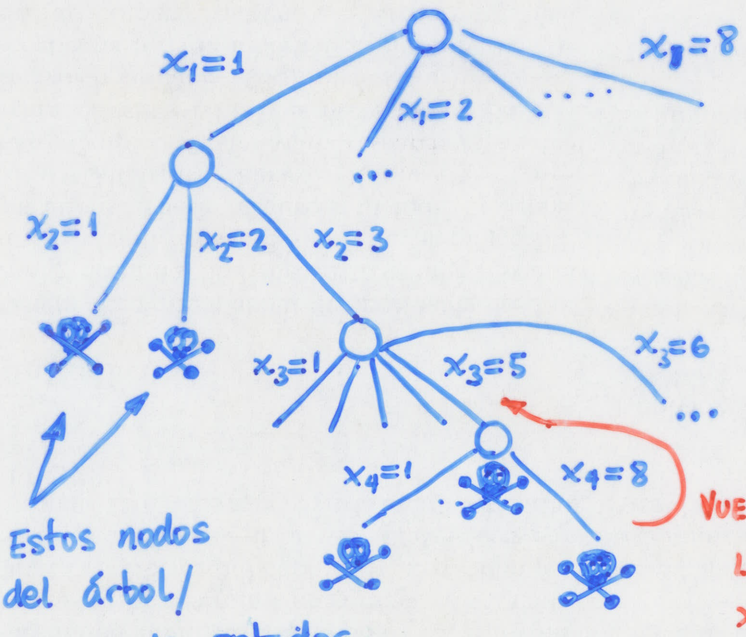
$(x_1, x_2, \dots, x_8)$ $x_i =$ COLUMNA PARA LA REINA SITUADA EN LA FILA i-ESIMA

$1 \leq x_i \leq 8$ ← RESTRICCION EXPLÍCITA

$\forall i,j : 1 \leq i < j \leq 8 : x_i \neq x_j$

$\forall i,j : 1 \leq i < j \leq 8 : |x_i - x_j| \neq j - i$

RESTRICCIONES IMPLÍCITAS



Estos nodos del árbol/ espacio de estados se PODAN

VUELTA ATRÁS! LA DECISIÓN $x_3=5$ SE DESHACE

BACKTRACKING

- SI EL ÁRBOL ES INDEPENDIENTE DE LA INSTANCIA A RESOLVER: ARBOL ESTÁTICO; EN CASO CONTRARIO: ÁRBOL DINÁMICO
- SI UN NODO GENERADO NO HA GENERADO TODOS SUS HIJOS SE DICE QUE ESTÁ VIVO
- EL NODO VIVO CUYOS HIJOS SE ESTÁN GENERANDO EN UN INSTANTE DADO SE DICE EN EXPANSIÓN
- UN NODO ESTÁ MUERTO SI TODOS SUS HIJOS HAN SIDO GENERADOS O NO SE EXPANDIRÁ MÁS (AUNQUE SEA completable*)
- SI UN NODO ESTADO NO ES COMPLETABLE NO SE EXPANDE; SE DICE QUE DICHO NODO HA SIDO PODADO

Potencia de backtracking (Ejemplo)

Número de 8-tuplas que satisfacen las restricciones implícitas =

$$8^8 = 16\,777\,216$$

"Fuerza bruta" encuentra la primera solución en 1.299.852 pasos

Número de 8-tuplas en filas y columnas distintas =

$$8! = 40.320$$

"Fuerza bruta" encuentra la 1ª solución en 2.830 pasos

⇨ Número de nodos en el árbol podado = 2057

⇨ Backtracking encuentra la 1ª solución en 114 pasos

* ESTO NO SUCEDE EN BACKTRACKING PURO PERO SÍ EN ALGUNAS VARIANTES

BACKTRACKING

- SI EL ÁRBOL ES INDEPENDIENTE DE LA INSTANCIA A RESOLVER: ARBOL ESTÁTICO; EN CASO CONTRARIO: ÁRBOL DINÁMICO

- SI UN NODO GENERADO NO HA GENERADO TODOS SUS HIJOS SE DICE QUE ESTÁ VIVO

- EL NODO VIVO CUYOS HIJOS SE ESTÁN GENERANDO EN UN INSTANTE DADO SE DICE EN EXPANSIÓN

- UN NODO ESTÁ MUERTO SI TODOS SUS HIJOS HAN SIDO GENERADOS O NO SE EXPANDIRÁ MÁS (AUNQUE SEA completable*)

- SI UN NODO ESTADO NO ES COMPLETABLE NO SE EXPANDE; SE DICE QUE DICHO NODO HA SIDO PODADO

Potencia de backtracking (Ejemplo)

Número de 8-tuplas que satisfacen las restricciones implícitas =

$$8^8 = 16\,777\,216$$

"Fuerza bruta" encuentra la primera solución en 1.299.852 pasos

Número de 8-tuplas en filas y columnas distintas =

$$8! = 40.320$$

"Fuerza bruta" encuentra la 1ª solución en 2.830 pasos

⇨ Número de nodos en el árbol podado = 2057

⇨ Backtracking encuentra la 1ª solución en 114 pasos

* ESTO NO SUCEDE EN BACKTRACKING PURO PERO SÍ EN ALGUNAS VARIANTES

BACKTRACKING

tupla de estado

nivel en curso

datos del problema

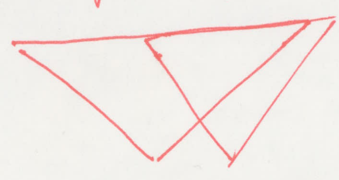
```
void backtracking(T x[], int k, T' z)

    prepara el recorrido del nivel k
    while (existe hermano de nivel k de x) {
        x = siguiente hermano de nivel k de x
        if (solucion(x, z))
            trata la solucion
        else if (completable(x, z))
            backtracking(x, k+1, z)
        else //poda
    }
```

Llamada inicial:

backtracking(x, 1, z)

BACKTRACKING

```
void    ocho_reinas(int x[], int k){

        x[k]=0;
        while (x[k] < 8) {
            x[k]= x[k]+ 1;
            if (k==8 && !amenaza(x, k))
                escribir_solucion(x, k)
            else if(!amenaza(x,k))
                backtracking(x, k+1)
        }

bool amenaza(int x[], int n) {

    for(i=1; i< n; i++)
        for(j= i+1; j≤ n; j++)
            if(x[i]=x[j] ||
                abs(x[i]-x[j])=j-i)
                    return true;
    return false;
}
```



BACKTRACKING

Para determinar si una solución es completable no hace falta rehacer todos los cálculos: sabemos que $x[1..k-1]$ es completable!

En general podemos "transmitir" información precomputada de un nodo estado x a sus hijos para mejorar la eficiencia. Esta técnica se denomina <u>marcaje</u>

```
bool amenaza(int x[], int k) {
    for (i=1; i<k; i++)
        if (x[i]= x[k] || abs(x[i]-x[k])= k-i)
            return true;
    return false;
}
```

BACKTRACKING

```
void backtracking(T x[], int k, T' z)

  if ( solucion(x, z))
      trata la solución

  {else} for ( x' = (x_1, ..., x_k, x'_{k+1}) completable )
              backtracking(x', k+1, z)
```

Se pone el <u>else</u> si y sólo si ningún nodo estado puede ser solución/respuesta, es decir, puede ser solución y "prefijo" de otra solución

En ciertas formulaciones p.e. del problema de la mochila todo nodo estado está en el espacio de soluciones

$$(x_1 = i_1,\ x_2 = i_2, \ldots,\ x_k = i_k)$$

$x_j = i_j \equiv$ el objeto i_j se coloca en la mochila

$k \equiv$ nº de objetos colocados en la mochila en la solución parcial

$1 \leq x_j \leq n$

BACKTRACKING

```
void backtracking(set<tupla> SOLUC, T' z) {

    int k; tupla x;

    k = 1;
    prepara el recorrido de nivel k
    while(k > 0) {
        if(existe hermano de nivel k) {
            x = siguiente hermano de nivel k de x
            if(solucion(x, z))
                SOLUC.insert(x);  // trata la solucion
            else if(completable(x, z)) {
                k++;
                prepara el recorrido de nivel k
            }
        } else { k--; }  // vuelta atrás
    }
}
```

BACKTRACKING

MARCAJE

En la versión recursiva:

- Mediante un parámetro adicional (inmersión de eficiencia)
- Se "marca" antes de la llamada recursiva, p.e. después de generar el nuevo nodo

  x= siguiente hermano de nivel k

  m = nueva_marca(m, x, k)

- Se "desmarca" al volver de la llamada recursiva, antes de intentar generar un nuevo nodo de nivel k

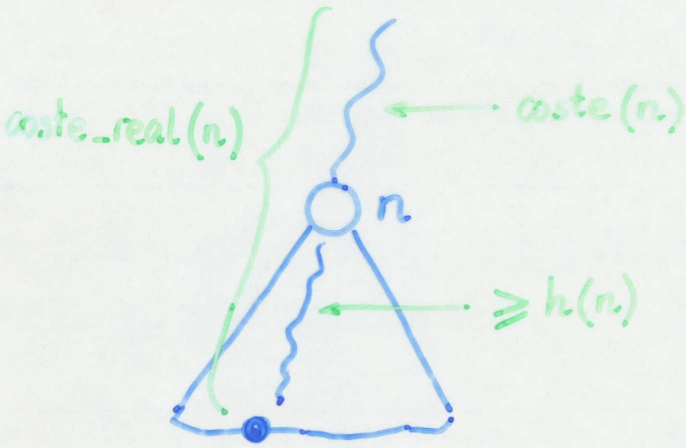
En la versión iterativa:

- Se crea un marcaje inicial m_0 antes de iniciar el bucle
- Se "marca" antes de bajar de nivel, de modo análogo a la versión recursiva
- Se "desmarca" al volver atrás

  ... else { k--; m = desmarcar(m, x, k); }

BACKTRACKING

SUPONGAMOS QUE BUSCAMOS LA SOLUCIÓN ÓPTIMA A UN PROBLEMA DE MINIMIZACIÓN Y QUE coste_mejor ES EL COSTE DE LA MEJOR SOLUCIÓN HALLADA HASTA EL MOMENTO



función de estimación o heurística

$$\text{coste\_estimado}(n) = \text{coste}(n) + h(n) =$$

una cota inferior al coste de la mejor solución alcanzable desde el nodo n,

i.e. $\leq$ coste_real(n)

BACKTRACKING

PODA BASADA EN LA MEJOR SOLUCIÓN EN CURSO (PBMSC)

si $coste\_estimado(n) > coste\_mejor$

entonces podamos (PBMSC) el nodo n !!

EL HEURÍSTICO $h(n)$ CUMPLE:

1) NO ENGAÑA: SI n ES COMPLETABLE,

$$h(n) \leq coste\_real(n) - coste(n)$$

2) ES "FÁCIL" DE CALCULAR

Para problemas de MINIMIZACIÓN SIEMPRE PODEMOS USAR $h(n) = 0$, SI LOS COSTES SON ≥ 0

En problemas de MAXIMIZACIÓN la estimación ha de ser siempre una cota superior al beneficio real

si $beneficio\_estimado(n) < beneficio\_mejor$

entonces PBMSC

BACKTRACKING

```
void mochila(int x[], double v[], double w[], double CAP,
             int n, int xopt[], double& vopt, int k,
             double pact, double vact) {
      ...
   x[k] = 2;        // prepara recorrido
   while (x[k] > 0) {     // mientras hay hermanos en el nivel k
      x[k]--;        // siguiente hermano de nivel k
      pact += w[k]*x[k]     // marcaje
      vact += v[k]*x[k]
      if (pact ≤ CAP)      // si es factible ...
         if (k==n)       // si es solucion ...
            if (vact > vopt) {     // si es mejor que la mejor
               xopt = x;        // solución hallada
               vopt = vact;
            }
         else              // no es solución, pero es factible ≡ completable
            if (estimacion(v,w,CAP,n,k,pact,vact)
               > vopt)    // si puede llevarnos a una solución mejor
             mochila(x,v,w,CAP,n,xopt,vopt,k+1,
                     pact,vact);
            else PBMSC
      else poda de nodo no completable
   }
}
```

BACKTRACKING

Antes de llamar a mochila(...) podemos obtener una solución inicial mediante el algoritmo voraz, descartando las fracciones

```
i = 1;  val = 0;  peso = 0;  xini = (0, ..., 0);
while (peso < CAP && i ≤ n)
    if (peso + w[i] ≤ CAP) {
        val += v[i];  xini[i] = 1;
        peso += w[i];
    }
```

mochila(x, v, w, CAP, n, xini, val, 1, 0, 0)

solución inicial (mejor en curso)

Esto permite actuar a la PBMSC desde el primer momento

BACKTRACKING

EL VALOR ALCANZABLE CON UNA SOLUCIÓN FRACCIONARIA AL PROBLEMA DE LA MOCHILA ES SIEMPRE $\geq$ QUE EL DE UNA SOLUCIÓN ENTERA. USAMOS ESTO COMO HEURÍSTICO

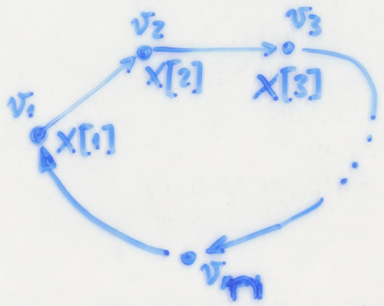
{ Pre: $(w[1], \dots, w[n]) = w$: pesos, $w[i] \geq 0$
$(v[1], \dots, v[n]) = v$: valores, $v[i] \geq 0$
$v[1]/w[1] \geq \dots \geq v[n]/w[n]$
CAP : capacidad, $CAP > 0$
pact, vact, k : peso y valor de la solución actual que involucra los 1..k primeros objetos }

```
double estimacion( ... ) {
    i = k; val = vact; peso = pact;
    while (peso < CAP && i < n) {
        i++;
        if ( peso + w[i] < CAP) {
            val += v[i];
            peso += w[i];
        }
        else { val += (CAP-peso)*v[i]/w[i];
               peso = CAP;
        }
    }
    return val;
}
```

BACKTRACKING

Ejemplo: Es hamiltoniano el grafo G?

X[i] = el i-ésimo vértice del ciclo



El primer vértice lo fijamos, y llamamos a

es_hamiltoniano:

```
X[1] = g.first_node(); for(i=1; i≤n; i++) X[i]= nil;
if(es_hamiltoniano(X, 2,g)){ ...
```

```
bool es_hamiltoniano(graph& g, node X[], int k){
    if (k== g.number_of_nodes()+1 &&
        x[k] == X[1])
        return true;
    else { do { ok= siguiente_tentativa(X, k); }
           while (ok && !es_hamiltoniano(g, X, k+1));
           return ok; // la tentativa era correcta ... y
                      // un ciclo hamiltoniano
         }
}
```

BACKTRACKING

Dado un camino $x_1, x_2, \dots, x_{k-1}, x_k$ reemplazar x_k por otro vértice x'_k tal que:

1) $(x_{k-1}, x'_k) \in E$ (sea una arista del grafo)

2) $x_k < x'_k$ (no queremos repetir tentativas)

3) si $k \neq n+1, x'_k \neq x_i \quad \forall i : 1 \leq i < k$ (no podemos cerrar el camino prematuramente)

```
ok = true;
while(ok) {
  if(X[k]==nil) X[k]=g.first_node();
  X[k]= g.succ_node(X[k]);
  if (X[k]==nil){ok=false; break;}
  for(i=1; i<k && X[i]!=X[k]; i++);
  if ((i==k || (i==1 && k==n+1)) &&
     es_arista(X[k-1], X[k], g))
     break;
}
return ok;
```

$\text{es\_arista}(u, v, g) \equiv (g.\text{adj\_nodes}(u)).\text{search}(v) \neq \text{nil}$

$v \in$ sucesores (u)

BACKTRACKING

Ejemplo: Dados n números positivos $w_1, w_2, \dots, w_n$ hallar las combinaciones cuya suma es igual a M.

Es parecido al problema de la mochila entera ...

REPRESENTAMOS UNA SOLUCION MEDIANTE $x = (x_1, \dots, x_n)$

$x_i = 0$ SI w_i APARECE EN LA COMBINACION

$x_i = 1$ SINO

- $x_i \in S_i$ ← EXPLICITA $S_i = \{0, 1\}$
- $\sum_{i=1}^{k} w_i \cdot x_i \leq M$ ← IMPLICITA
- $\sum_{i=1}^{k} w_i \cdot x_i + \sum_{i=k+1}^{n} w_i \geq M \equiv$ COMPLETABLE

USAREMOS EL MARCAJE PARA "RECORDAR" LA SUMA DE UNA SOLUCION PARCIAL

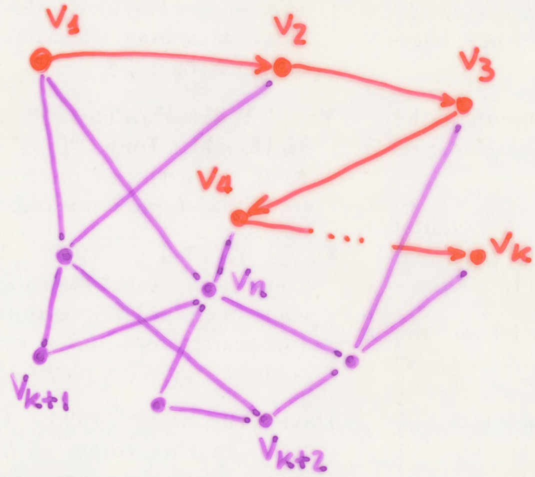
Y REORDENAREMOS EL NIVEL k PARA EMPEZAR PROBANDO $x_k = 1$ PUES PROBABLEMENTE NOS PERMITIRÁ PODAR ANTES

BACKTRACKING

```
void subsum(int w[], int M, int n) {

    k=1;  s=0;  r = Σ_{i=1}^{n} w[i];
    x[1] = 2;  nsol = 0;
                                                  marcaje
    while (k>0) {
        if (x[k]>0) {
            x[k] = x[k] - 1;  s = s + w[k] * x[k];  r = r - w[k];
            if (k==n && s==M) {
                imprimir_solucion(x, n, w, M);
                nsol++;
            }
            else if (s+r >= M) {
                    k++;  x[k] = 2;
            }                                     desmarcaje
        else { s = s - w[k] * x[k];  r = r + w[k];
            k--;
        }
    }
    printf("Soluciones halladas: %d\n", nsol);

}
```

BACKTRACKING

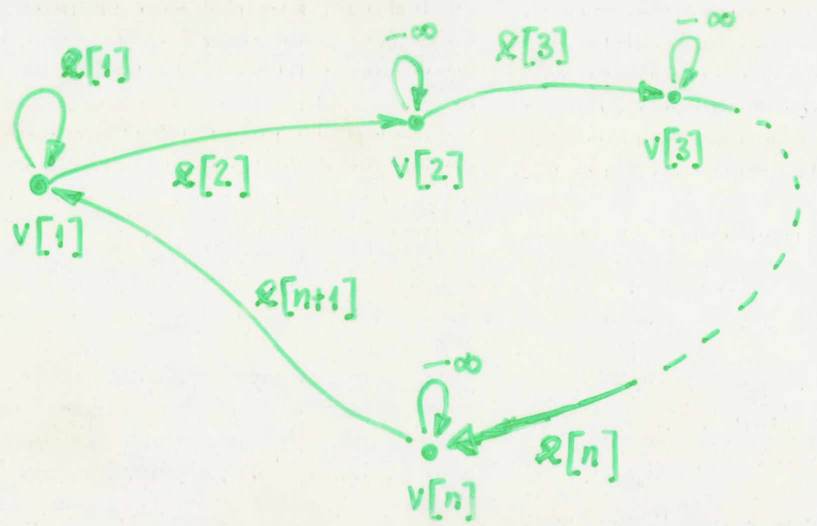
Dado un grafo no dirigido y conexo, etiquetado con pesos/distancias en $\mathbb{R}^+$, hallar un ciclo hamiltoniano en G de peso mínimo.



$h(n) = ?$

$\Rightarrow$ Si construímos un árbol de expansión mínimo para el subgrafo inducido por $\{v_{k+1}, \dots, v_n\}$ su peso/coste total será inferior al de un camino que pase por $v_{k+1}, \dots, v_n$ exactamente una vez

BACKTRACKING



e[i] = arco de v[i-1] a v[i] en el camino en curso

v[i] = vértice i-ésimo del camino en curso

cost[e] = coste del arco e (≥ 0)

cost[e[1]] = $-\infty$; e[1] es un arco "ficticio" que simplifica la implementación

También hay arcos (v[i], v[i]) de coste $-\infty$ cuya finalidad es preparar los recorridos de cada nivel

BACKTRACKING

```
void TSP(graph& g, edge_array<double>& cost)
{   ...
```

obtenemos una solución inicial ([illegible]

```
    forall_nodes(v, g) {
        a = g.new_edge(v, v);
        cost[a] = -∞;
    }
    g.sort_edges(cost);

    v[1] = g.first_node();
    e[1] = g.first_adj_edge(v[1]);

    a = e[1];  k = 2;     ← preparamos el recorrido
    cost_act = 0;   ← marcaje
```

BACKTRACKING

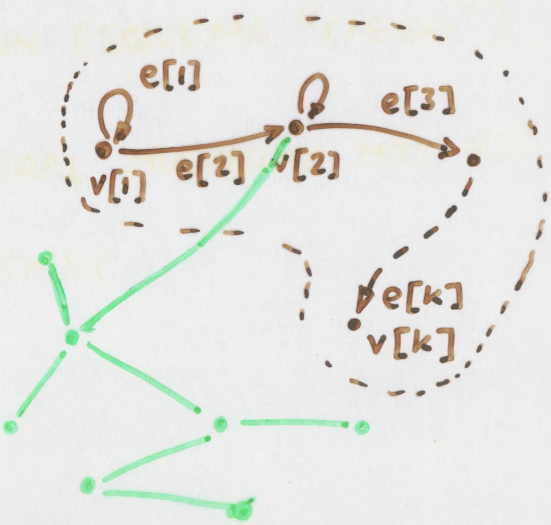
```
...

while ( k > 1 ) {

  if ( (a= g.adj_succ(a)) ≠ nil ) {

      v[k]= g.target (a);
      e[k]= a;
      cost_act += cost[a];            ← marcaje
      if ( factible(g, v, e, k))

         if (k== n+1 && cost_act < coste_mejor) {

               emejor = e;   vmejor = v;
               coste_mejor = cost_act;           DESMARCAR
         }     cost_act -= cost[a];
         else if (k < n+1 && estim(g, v, e, k, cost_act)
                                 < coste_mejor) {

                 a= g.first_adj_edge(v[k]);
                 k++;
         }
         else {//PBMSC ...           hay que DESMARCAR
   else {//nodo no completable                       desmarcaje
           cost_act -= cost[a];}
  } else {  cost_act -= cost[e[k]];
            k--;      ← vuelta atrás
         }

}

}
```

BACKTRACKING

```
bool factible( ... ) {

    if (k ≤ n)
        return !repetido(g, v, k);
    else
        return v[n+1] == v[1];
}

bool repetido( ... ) {

    for (i=1; i<k; i++)
        if (v[i]== v[k]) break;
    return i<k;
}
```

BACKTRACKING

```
double estim( ... ) {

  node_partition P(g); ...

   for (i = 2; i ≤ k; i++)
       P. union_blocks(v[1], v[i]);
   coste= cost_act;
   forall_edges (a, g) {
      u= g. source(a);
      w= g.target(a);
      if (u ≠ w && ! P. same_block(u,w)) {
          P.union_blocks(u,w);
          coste += cost[a];
      }
   }
   return coste;
}
```



$$coste = \sum_{i=2}^{k} cost[e[i]] + coste(T)$$