Ramificación y poda

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Enero 2016

Bibliografía

- R. Neapolitan y K. Naimipour. *Foundations of Algorithms using C++ pseudocode*. Tercera edición. Jones and Bartlett Publishers, 2004.
  Capítulo 6
- E. Horowitz, S. Sahni y S. Rajasekaran. *Computer Algorithms*. Tercera edición. Computer Science Press, 1998.
  Capítulo 8
- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición, Garceta, 2013.
  Capítulo 15

Búsqueda en el espacio de soluciones

- No siempre podemos utilizar técnicas como divide y vencerás, método voraz o programación dinámica para lograr soluciones *eficientes*.
- El último recurso que nos queda para resolver nuestro problema es aplicar la "fuerza bruta".
- Realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos.

Búsqueda en el espacio de soluciones

- No siempre podemos utilizar técnicas como divide y vencerás, método voraz o programación dinámica para lograr soluciones *eficientes*.
- El último recurso que nos queda para resolver nuestro problema es aplicar la "fuerza bruta".
- Realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos.
- Esta búsqueda exhaustiva resultará impracticable si el cardinal del conjunto de soluciones posibles es muy grande, lo cual ocurre muy a menudo.
- Es imprescindible intentar estructurar el espacio a explorar, tratando de descartar en bloque posibles soluciones no satisfactorias.

Búsqueda en el espacio de soluciones

- Consideraremos problemas cuyas soluciones sean construibles por etapas.
- Una solución será expresable en forma de n-tupla $(x_1, \ldots, x_n)$, donde cada $x_i \in S_i$ representa la decisión tomada en la etapa i-ésima.

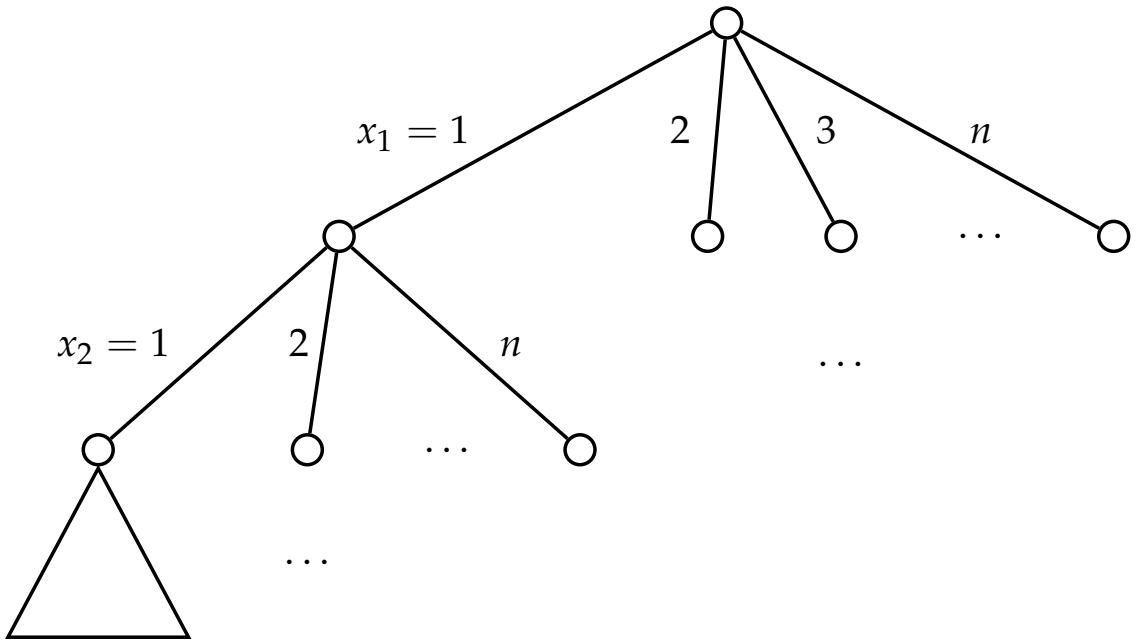
Búsqueda en el espacio de soluciones

- Consideraremos problemas cuyas soluciones sean construibles por etapas.
- Una solución será expresable en forma de n-tupla $(x_1, \ldots, x_n)$, donde cada $x_i \in S_i$ representa la decisión tomada en la etapa i-ésima.
- Además, una solución habrá de minimizar, maximizar o simplemente satisfacer una cierta función criterio.
- Se establecen dos categorías de restricciones:
  - las restricciones explícitas, que indican los conjuntos S_i a los que pertenecen cada una de las componentes de una tupla solución;
  - las restricciones implícitas, que son las relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la función criterio.

Búsqueda en el espacio de soluciones

- Consideraremos problemas cuyas soluciones sean construibles por etapas.
- Una solución será expresable en forma de n-tupla $(x_1, \ldots, x_n)$, donde cada $x_i \in S_i$ representa la decisión tomada en la etapa i-ésima.
- Además, una solución habrá de minimizar, maximizar o simplemente satisfacer una cierta función criterio.
- Se establecen dos categorías de restricciones:
  - las restricciones explícitas, que indican los conjuntos S_i a los que pertenecen cada una de las componentes de una tupla solución;
  - las restricciones implícitas, que son las relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la función criterio.
- El espacio de soluciones estará formado por el conjunto de tuplas que satisfacen las restricciones explícitas, y se puede estructurar como un árbol de exploración.

Búsqueda en el espacio de soluciones



Búsqueda en el espacio de soluciones

- Se utilizan funciones de poda o tests de factibilidad que permiten determinar cuándo una tupla parcial nunca va a conducir a una solución satisfactoria, por lo que es inútil seguir buscando a partir de ella.
- Una vez definido el árbol de exploración, el algoritmo para resolver el problema consistirá en realizar un recorrido del árbol en cierto orden, hasta encontrar la primera solución, o bien recorrer el árbol completo (salvo las zonas podadas) para obtener todas las soluciones o la solución óptima deseada.
- Durante el proceso, para cada nodo se irán generando sus sucesores; así denominaremos nodos vivos a aquellos para los cuales todavía no se han generado todos sus hijos; nodos en expansión a aquellos cuyos hijos están siendo generados; y nodos muertos a aquellos que no pueden ser expandidos, bien porque no hayan superado el test de factibilidad, o bien porque todos sus hijos ya han sido generados.

Búsqueda en el espacio de soluciones

- Existen dos tipos de recorridos:

  Vuelta atrás: el recorrido se realiza en profundidad, de forma que los nodos vivos se gestionan mediante una pila. El método resulta sencillo y eficiente en espacio.

  Ramificación y poda: corresponde a una búsqueda más "inteligente", donde en cada momento se expande el nodo vivo más "prometedor", de forma que los nodos vivos se gestionan mediante una cola con prioridad.

Búsqueda en el espacio de soluciones

- Existen dos tipos de recorridos:

  Vuelta atrás: el recorrido se realiza en profundidad, de forma que los nodos vivos se gestionan mediante una pila. El método resulta sencillo y eficiente en espacio.

  Ramificación y poda: corresponde a una búsqueda más "inteligente", donde en cada momento se expande el nodo vivo más "prometedor", de forma que los nodos vivos se gestionan mediante una cola con prioridad.

- El coste en el caso peor de todos estos algoritmos de búsqueda exhaustiva está en el orden del tamaño del espacio de soluciones, que suele ser cuando menos exponencial. Su utilidad práctica reside en la efectividad de las funciones de poda que se apliquen: cuanto más elaboradas sean, más nodos no factibles se detectarán y mayor proporción de árbol se podará. Sin embargo, si se vuelven demasiado sofisticadas, su coste de aplicación (a cada nodo del árbol) puede no llegar a compensar la poda obtenida.

Esquema general de vuelta atrás

Cuando se realiza una búsqueda en profundidad y se llega a un nodo muerto, hay que deshacer la última decisión tomada, para optar por la siguiente alternativa (como cuando en un laberinto se llega a un callejón sin salida).

```
proc vuelta-atrás(sol : tupla, e k : nat)
    preparar-recorrido-nivel(k)
    mientras ¬último-hijo-nivel(k) hacer
        sol[k] := siguiente-hijo-nivel(k)
        si es-solución?(sol,k) entonces
            tratar-solución(sol)
        si no
            si es-completable?(sol,k) entonces
                vuelta-atrás(sol,k+1)
            fsi
        fsi
    fmientras
fproc
```

Vuelta atrás con marcaje

Se puede optimizar el test de factibilidad por el método de asociar a cada nodo cierta información que corresponde a "cálculos parciales" de dichos tests.

Se utilizan parámetros adicionales (marcadores) de entrada/salida que equivalen a variables globales y por tanto suponen un pequeño incremento del coste en espacio.

```
proc vuelta-atrás-marcaje(sol : tupla, e k : nat, m : marcador)
    preparar-recorrido-nivel(k)
    mientras ¬último-hijo-nivel(k) hacer
        sol[k] := siguiente-hijo-nivel(k)
        m := marcar(m, sol[k])
        si es-solución?(sol, k) entonces
            tratar-solución(sol)
        si no
            si es-completable?(sol, k, m) entonces
                vuelta-atrás-marcaje(sol, k + 1, m)
            fsi
        fsi
        m := desmarcar(m, sol[k])
    fmientras
fproc
```

Problemas de optimización

- Se modifican los esquemas anteriores de forma que se almacene la mejor solución encontrada hasta el momento.
- Al tratar una nueva solución, se comparará con la que se tiene almacenada.
- Para realizar esta comparación de la forma más eficiente posible, es preciso almacenar junto con la mejor solución su valor asociado.
- Además, para facilitar el cálculo del valor de cada solución alcanzada se incorpora como marcador el valor (parcial) de la tupla parcial.
- Los problemas de optimización admiten un mecanismo adicional de poda, que se realiza cuando se puede asegurar que ninguno de los descendientes del nodo a expandir puede llegar a alcanzar una solución mejor que la mejor encontrada hasta ese momento.
- En un problema de minimización, una cota inferior o estimación de la mejor solución alcanzable desde un nodo sirve para podarlo si la estimación es ya mayor que el valor asociado a la mejor solución encontrada hasta el momento.

Esquema de optimización

```
proc vuelta-atrás-optimización(sol : tupla, e k : nat, valor : valor,
                               sol-mejor : tupla, valor-mejor : valor)
    preparar-recorrido-nivel(k)
    mientras ¬último-hijo-nivel(k) hacer
        sol[k] := siguiente-hijo-nivel(k)
        valor := actualizar(valor, sol, k)
        si es-solución?(sol, k) entonces
            si mejor(valor, valor-mejor) entonces
                sol-mejor := sol ; valor-mejor := valor
            fsi
        si no
            si es-completable?(sol, k) ∧ es-prometedor?(sol, k, valor, valor-mejor) entonces
                vuelta-atrás-optimización(sol, k + 1, valor, sol-mejor, valor-mejor)
            fsi
        fsi
        valor := desactualizar(valor, sol, k)
    fmientras
fproc
```

Ramificación y poda

- Ramificación y poda es otra técnica de exploración de espacios de soluciones.
- La diferencia esencial con respecto a vuelta atrás es la forma de recorrer el espacio de soluciones.
- Aquí la gestión de los nodos vivos se realiza mediante una cola con prioridad, expandiendo en cada momento el nodo vivo "más prometedor".

Ramificación y poda

- Ramificación y poda es otra técnica de exploración de espacios de soluciones.
- La diferencia esencial con respecto a vuelta atrás es la forma de recorrer el espacio de soluciones.
- Aquí la gestión de los nodos vivos se realiza mediante una cola con prioridad, expandiendo en cada momento el nodo vivo "más prometedor".
- La variante más utilizada de ramificación y poda es aquella que busca la solución óptima de entre todas las posibles soluciones al problema, esperando encontrarla de forma más rápida que por vuelta atrás.

Ramificación y poda

- Ramificación y poda es otra técnica de exploración de espacios de soluciones.
- La diferencia esencial con respecto a vuelta atrás es la forma de recorrer el espacio de soluciones.
- Aquí la gestión de los nodos vivos se realiza mediante una cola con prioridad, expandiendo en cada momento el nodo vivo "más prometedor".
- La variante más utilizada de ramificación y poda es aquella que busca la solución óptima de entre todas las posibles soluciones al problema, esperando encontrarla de forma más rápida que por vuelta atrás.
- Las características de estos problemas de optimización:
  - la solución es expresable en forma de tupla $(x_1, \ldots, x_n)$,
  - es posible determinar si una tupla es una solución factible,
  - es posible determinar si una tupla parcial $(x_1, \ldots, x_k)$ puede ser completada hasta una solución factible.

Ramificación y poda (minimización)

- Además, es necesario disponer de una función `coste-estimado` que, dada una tupla parcial $X = (x_1, \ldots, x_k)$, proporcione una cota inferior al coste de la mejor solución alcanzable desde X.
- Si llamamos `coste-real`(X) al coste de dicha mejor solución, entonces ha de cumplirse que

$$\texttt{coste-estimado}(X) \leq \texttt{coste-real}(X).$$

Ramificación y poda (minimización)

- Además, es necesario disponer de una función `coste-estimado` que, dada una tupla parcial $X = (x_1, \ldots, x_k)$, proporcione una cota inferior al coste de la mejor solución alcanzable desde X.
- Si llamamos `coste-real`(X) al coste de dicha mejor solución, entonces ha de cumplirse que

$$\texttt{coste-estimado}(X) \leq \texttt{coste-real}(X).$$

- Si `coste-estimado`(X) se aproxima suficientemente al valor del coste real, es de esperar que los nodos con un bajo coste estimado conducirán a soluciones con un bajo coste real, y por eso consideraremos como más prometedor aquel nodo vivo que tenga un coste estimado mínimo.

Ramificación y poda (minimización)

- Además, es necesario disponer de una función `coste-estimado` que, dada una tupla parcial $X = (x_1, \ldots, x_k)$, proporcione una cota inferior al coste de la mejor solución alcanzable desde X.
- Si llamamos `coste-real`(X) al coste de dicha mejor solución, entonces ha de cumplirse que

$$\texttt{coste-estimado}(X) \leq \texttt{coste-real}(X).$$

- Si `coste-estimado`(X) se aproxima suficientemente al valor del coste real, es de esperar que los nodos con un bajo coste estimado conducirán a soluciones con un bajo coste real, y por eso consideraremos como más prometedor aquel nodo vivo que tenga un coste estimado mínimo.
- Este coste estimado, como en vuelta atrás, proporciona una forma adicional de poda. Si el coste estimado no es menor que el coste de la mejor solución obtenida hasta el momento, no merece la pena seguir explorando esa rama del árbol.

Esquema general de minimización

fun ramificación-y-poda-mín(T : *árbol-de-estados*) **dev** $\langle$ *sol-mejor* : *tupla*, *coste-mejor* : *valor* $\rangle$
var X, Y : *nodo*, C : *colapr[nodo]*
    $Y :=$ raíz(T)
    $C :=$ cp-vacía() ; añadir(C, Y)
    coste-mejor $:= +\infty$
    mientras $\neg$es-cp-vacía?(C) $\wedge$ coste-estimado(mínimo(C)) $<$ *coste-mejor* **hacer**
        $Y :=$ mínimo(C) ; eliminar-mín(C)
        para todo hijo X de Y **hacer**
            si es-solución?(X) **entonces**
                si coste-real(X) $<$ *coste-mejor* **entonces**
                    coste-mejor $:=$ coste-real(X) ; *sol-mejor* $:=$ solución(X)
                fsi
            si no
                si es-completable?(X) $\wedge$ coste-estimado(X) $<$ *coste-mejor* **entonces**
                    añadir(C, X)
                fsi
            fsi
        fpara
    fmientras
ffun

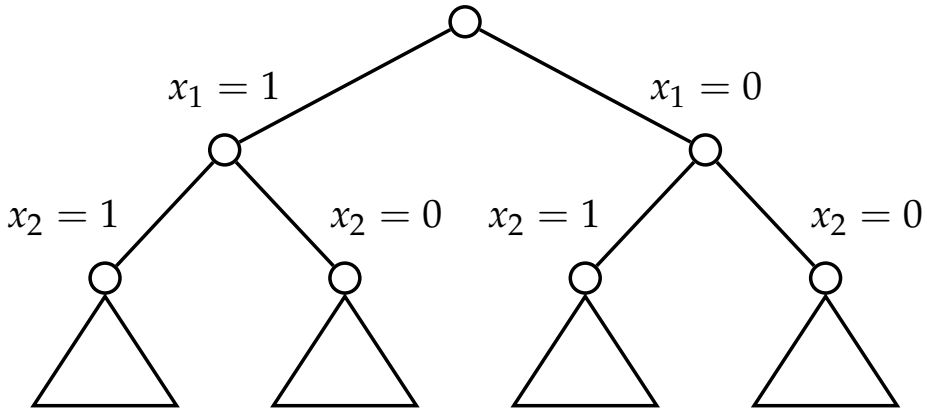
Tareas con plazo fijo, duración y coste

Tenemos n tareas y un procesador. Cada tarea i tiene asociada una terna (t_i, p_i, c_i) que indica que la tarea i tarda un tiempo t_i en ejecutarse, y que si su ejecución no está completa antes del plazo p_i, deberá pagarse una multa c_i por ella.

El objetivo es seleccionar un subconjunto S de las n tareas tal que las tareas en S se puedan realizar antes de su plazo y que la multa impuesta por las tareas que no están en S (no realizadas) sea mínima.

Tareas con plazo fijo, duración y coste

- El subconjunto de tareas realizadas se representa mediante su función característica, es decir, mediante una tupla $(x_1, \ldots, x_n)$ donde $x_i = 1$ indica que la tarea i-ésima se realiza, mientras que $x_i = 0$ indica que no se realiza.
- Se obtiene un árbol de exploración binario.



Tareas con plazo fijo, duración y coste

- Para comprobar la factibilidad del subconjunto seleccionado se puede utilizar la siguiente propiedad: S es factible si y solo si la secuencia que ordena las tareas en S de forma no decreciente según plazo es admisible.

Tareas con plazo fijo, duración y coste

- Para comprobar la factibilidad del subconjunto seleccionado se puede utilizar la siguiente propiedad: S es factible si y solo si la secuencia que ordena las tareas en S de forma no decreciente según plazo es admisible.
- En cuanto a la estimación, necesitamos una cota inferior:
  1. Coste actual, $\sum_{i=1}^{k}(1-x_i)c_i$.
  2. Coste actual más el coste de las tareas pendientes (desde $k+1$ hasta n) que ya no pueden hacerse antes de su plazo.

Tareas con plazo fijo, duración y coste

```
struct Tarea {
    float t, p, c;
};

struct Nodo {
    vector<bool> sol;
    int k;
    float tiempo; // tiempo ocupado
    float coste;  // coste acumulado
    float coste_est; // prioridad
};

// para ordenar en la cola con prioridad
bool operator<(Nodo const& a, Nodo const& b) {
    return a.coste_est < b.coste_est;
}
```

Tareas con plazo fijo, duración y coste

```
float calculo_estimacion(vector<Tarea> const& tareas, int k,
                         float tiempo, float coste) {
    float est = coste;
    for (int i = k+1; i < tareas.size(); ++i) {
        if (tiempo + tareas[i].t > tareas[i].p) {
            est += tareas[i].c;
        }
    }
    return est;
}
```

Tareas con plazo fijo, duración y coste

```
// tareas ordenadas de menor a mayor plazo
void tareas_rp(vector<Tarea> const& tareas,
               vector<bool> & sol_mejor, float & coste_mejor) {
    size_t N = tareas.size();
    // se genera la raíz
    Nodo Y;
    Y.sol = vector<bool>(N, false);
    Y.k = -1;
    Y.tiempo = Y.coste = 0;
    Y.coste_est = calculo_estimacion(tareas, Y.k, Y.tiempo, Y.coste);
    PriorityQueue<Nodo> cola;
    cola.push(Y);
    coste_mejor = numeric_limits<float>::infinity();
    // búsqueda mientras pueda haber algo mejor
    while (!cola.empty() && cola.top().coste_est < coste_mejor) {
        Y = cola.top(); cola.pop();
        Nodo X(Y);
        ++X.k;
```

Tareas con plazo fijo, duración y coste

```
            // hijo izquierdo, realizar la tarea
            if (Y.tiempo + tareas[X.k].t <= tareas[X.k].p) {
                X.sol[X.k] = true; X.tiempo = Y.tiempo + tareas[X.k].t;
                X.coste = Y.coste;
                X.coste_est = calculo_estimacion(tareas, X.k, X.tiempo, X.coste);
                if (X.coste_est < coste_mejor) {
                    if (X.k == N-1) {
                        sol_mejor = X.sol; coste_mejor = X.coste;
                    } else { cola.push(X); }
                }
            }
            // hijo derecho, no realizar la tarea
            X.sol[X.k] = false; X.tiempo = Y.tiempo;
            X.coste = Y.coste + tareas[X.k].c;
            X.coste_est = calculo_estimacion(tareas, X.k, X.tiempo, X.coste);
            if (X.coste_est < coste_mejor) {
                if (X.k == N-1) {
                    sol_mejor = X.sol; coste_mejor = X.coste;
                } else { cola.push(X); }
            }
        }
    }
```

Funcionarios con trabajo

El Ministro de Desinformación y Decencia se ha propuesto hacer trabajar en firme a sus n funcionarios, para lo que se ha sacado de la manga n trabajos.

A pesar de su ineficacia, todos los funcionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros.

La información al respecto se recoge en la tabla $T[1..n, 1..n]$, donde $T[i,j]$ representa el tiempo que el funcionario i tarda en realizar el trabajo j.

Para justificar su puesto, Su Excelencia el Sr. Ministro desea conocer la asignación óptima de trabajos a funcionarios de modo que la suma total de tiempos sea mínima.

Funcionarios con trabajo

Podemos representar la solución en tuplas de la forma $(x_1, x_2, \ldots, x_n)$ donde x_i es el trabajo asignado al funcionario i.

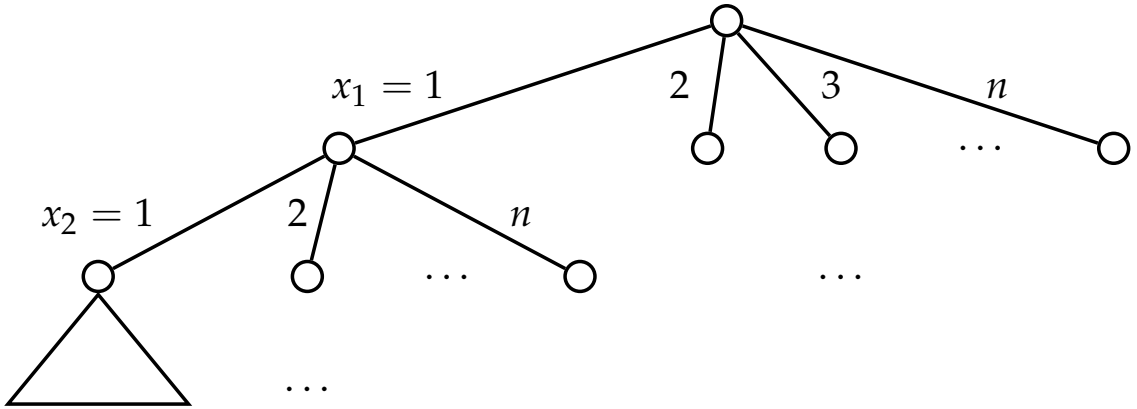
Las soluciones tienen que cumplir que son permutaciones de los n trabajos.

Funcionarios con trabajo

Podemos representar la solución en tuplas de la forma $(x_1, x_2, \ldots, x_n)$ donde x_i es el trabajo asignado al funcionario i.

Las soluciones tienen que cumplir que son permutaciones de los n trabajos.

El árbol de exploración, donde cada nodo tiene n hijos y hay n niveles, es

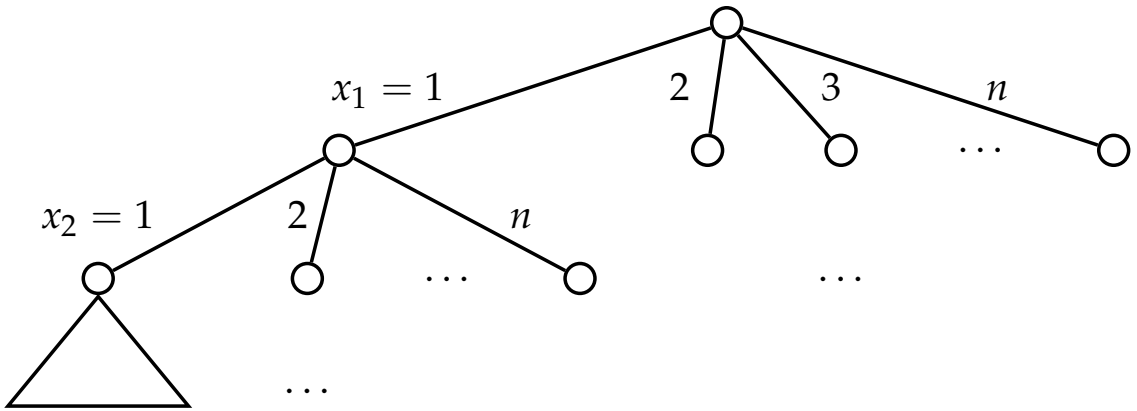


Funcionarios con trabajo

Podemos representar la solución en tuplas de la forma $(x_1, x_2, \ldots, x_n)$ donde x_i es el trabajo asignado al funcionario i.

Las soluciones tienen que cumplir que son permutaciones de los n trabajos.

El árbol de exploración, donde cada nodo tiene n hijos y hay n niveles, es



Ya que cada trabajo solo puede ser asignado a un funcionario, llevaremos cuenta de los trabajos ya asignados en un vector marcador $asignado[1..n]$ de booleanos.

Elegir los x_i tales que se minimice $\sum_{i=1}^{n} T[i, x_i]$.

Funcionarios con trabajo: cota

Veamos cómo podemos obtener una cota inferior del coste total a partir del coste de una solución parcial.

Para la solución parcial $(x_1, \ldots, x_k)$, el tiempo hasta el momento es $tiempo = \sum_{i=1}^{k} T[i, x_i]$, y tenemos que estimar el tiempo del resto de la solución.

Funcionarios con trabajo: cota

Veamos cómo podemos obtener una cota inferior del coste total a partir del coste de una solución parcial.

Para la solución parcial $(x_1, \ldots, x_k)$, el tiempo hasta el momento es $tiempo = \sum_{i=1}^{k} T[i, x_i]$, y tenemos que estimar el tiempo del resto de la solución.

1. La opción más sencilla (y más optimista) es aproximar con 0 este tiempo, y utilizar *tiempo* como estimación.

Funcionarios con trabajo: cota

Veamos cómo podemos obtener una cota inferior del coste total a partir del coste de una solución parcial.

Para la solución parcial $(x_1, \ldots, x_k)$, el tiempo hasta el momento es $tiempo = \sum_{i=1}^{k} T[i, x_i]$, y tenemos que estimar el tiempo del resto de la solución.

1. La opción más sencilla (y más optimista) es aproximar con 0 este tiempo, y utilizar *tiempo* como estimación.
2. Otra posibilidad es calcular un mínimo global de la matriz T,

$$mínT = \text{mín}\{T[i,j] \mid 1 \leq i \leq n \ \wedge \ 1 \leq j \leq n\},$$

que sirva como cota inferior al tiempo de realización de cada trabajo por los funcionarios.
Así podemos aproximar el tiempo del resto de la solución con $(n-k)\, mínT$.

Funcionarios con trabajo: cota

3. Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$\mathit{rápido}[i] = \mathbf{mín}\{T[i,j] \mid 1 \leq j \leq n\}.$$

De esta manera podemos calcular la estimación del tiempo restante como:

$$\sum_{i=k+1}^{n} \mathit{rápido}[i].$$

Funcionarios con trabajo: cota

3. Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$\mathit{rápido}[i] = \mathbf{mín}\{T[i,j] \mid 1 \leq j \leq n\}.$$

De esta manera podemos calcular la estimación del tiempo restante como:

$$\sum_{i=k+1}^{n} \mathit{rápido}[i].$$

4. Calcular *rápido* dinámicamente entre los trabajos no repartidos ya:

$$\mathit{rápido\text{-}din}[i] = \mathbf{mín}\{T[i,j] \mid 1 \leq j \leq n \ \wedge \ j \text{ no ha sido asignado}\}.$$

Funcionarios con trabajo: cota

3. Tener calculado un mínimo por cada fila: para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

$$\mathit{rápido}[i] = \mathbf{mín}\{T[i,j] \mid 1 \leq j \leq n\}.$$

De esta manera podemos calcular la estimación del tiempo restante como:

$$\sum_{i=k+1}^{n} \mathit{rápido}[i].$$

4. Calcular *rápido* dinámicamente entre los trabajos no repartidos ya:

$$\mathit{rápido\text{-}din}[i] = \mathbf{mín}\{T[i,j] \mid 1 \leq j \leq n \ \wedge \ j \text{ no ha sido asignado}\}.$$

Implementamos la opción 3. Además calcularemos inicialmente las sumas

$$\mathit{est}[k] = \sum_{i=k+1}^{n} \mathit{rápido}[i]$$

de tal forma que no tengan que ser recalculados dentro del algoritmo.

Problema de la mochila

Cuando Alí-Babá consigue por fin entrar en la Cueva de los Cuarenta Ladrones encuentra allí gran cantidad de objetos muy valiosos. A pesar de su pobreza, Alí-Babá conoce muy bien el peso y valor de cada uno de los objetos en la cueva.

Debido a los peligros que tiene que afrontar en su camino de vuelta, solo puede llevar consigo aquellas riquezas que quepan en su pequeña mochila, que soporta un peso máximo conocido.

Determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

Problema de la mochila

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n.

La mochila soporta un peso total máximo $M > 0$.

Problema de la mochila

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n.

La mochila soporta un peso total máximo $M > 0$.

El problema consiste en maximizar

$$\sum_{i=1}^{n} x_i v_i$$

con la restricción

$$\sum_{i=1}^{n} x_i p_i \leq M,$$

donde $x_i \in \{0, 1\}$ indica si hemos cogido (1) o no (0) el objeto i.

Problema de la mochila

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n.

La mochila soporta un peso total máximo $M > 0$.

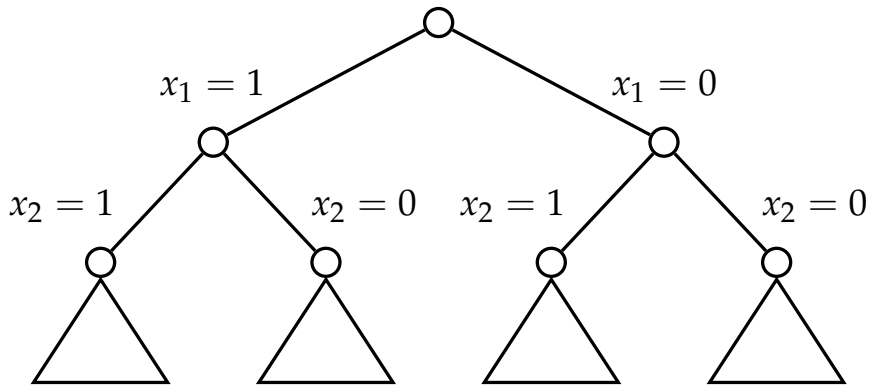
El problema consiste en maximizar

$$\sum_{i=1}^{n} x_i v_i$$

con la restricción

$$\sum_{i=1}^{n} x_i p_i \leq M,$$

donde $x_i \in \{0, 1\}$ indica si hemos cogido (1) o no (0) el objeto i.



Problema de la mochila: cota

- Como el problema es de maximización, tenemos que encontrar una cota superior de la mejor solución alcanzable.

Problema de la mochila: cota

- Como el problema es de maximización, tenemos que encontrar una cota superior de la mejor solución alcanzable.
- Utilizamos el algoritmo voraz que resolvía este mismo problema cuando los objetos se podían fraccionar ($0 \leq x_i \leq 1$).
- Ya que esta solución era óptima, no puede haber ninguna solución sin fraccionar objetos que sea mejor.
- Para utilizar este algoritmo necesitamos que los objetos estén ordenados en orden decreciente de valor por unidad de peso, v_i/p_i.

Problema de la mochila: implementación

```
struct Objeto {
    float peso;
    float valor;
};

struct Nodo {
    vector<bool> sol;
    int k;
    float peso; // peso acumulado
    float beneficio; // valor acumulado
    float benef_est; // prioridad
};

bool operator>(Nodo const& a, Nodo const& b) {
    return a.benef_est > b.benef_est;
}
```

Problema de la mochila: implementación

```
// estrategia voraz, para calcular la estimación
float calculo_voraz(vector<Objeto> const& objetos, float M, int k,
                    float peso, float beneficio) {
    float hueco = M - peso;
    float estimacion = beneficio;
    int j = k+1;
    while (j < objetos.size() && objetos[j].peso <= hueco) {
        // podemos coger el objeto j entero
        hueco -= objetos[j].peso;
        estimacion += objetos[j].valor;
        ++j;
    }
    if (j < objetos.size()) {
        estimacion += (hueco / objetos[j].peso) * objetos[j].valor;
    }
    return estimacion;
}
```

Problema de la mochila: implementación

```
// objetos ordenados de mayor a menor valor/peso
void mochila_rp(vector<Objeto> const& objetos, float M,
                vector<bool> & sol_mejor, float & benef_mejor) {
    size_t N = objetos.size();
    // se genera la raíz
    Nodo Y;
    Y.sol = vector<bool>(N);
    Y.k = -1;
    Y.peso = Y.beneficio = 0;
    Y.benef_est = calculo_voraz(objetos, M, Y.k, Y.peso, Y.beneficio);
    PriorityQueue<Nodo,std::greater<Nodo>> cola;
    cola.push(Y);
    benef_mejor = -1;
    // búsqueda mientras pueda haber algo mejor
    while (!cola.empty() && cola.top().benef_est > benef_mejor) {
        Y = cola.top(); cola.pop();
        Nodo X(Y);
        ++X.k;
```

Problema de la mochila: implementación

```
            // probamos a meter el objeto en la mochila
            if (Y.peso + objetos[X.k].peso <= M) {
                X.sol[X.k] = true;
                X.peso = Y.peso + objetos[X.k].peso;
                X.beneficio = Y.beneficio + objetos[X.k].valor;
                X.benef_est = Y.benef_est;
                if (X.k == N-1) {
                    sol_mejor = X.sol; benef_mejor = X.beneficio;
                } else { cola.push(X); }
            }
            // probar a no meter el objeto
            X.benef_est = calculo_voraz(objetos, M, X.k, Y.peso, Y.beneficio);
            if (X.benef_est > benef_mejor) {
                X.peso = Y.peso; X.beneficio = Y.beneficio;
                X.sol[X.k] = false;
                if (X.k == N-1) {
                    sol_mejor = X.sol; benef_mejor = X.beneficio;
                } else { cola.push(X); }
            }
        }
    }
```

Problema del viajante

Un vendedor ambulante de alfombras persas tiene que recorrer n ciudades volviendo tras ello al punto de partida. El buen señor se ha informado sobre las posibles conexiones directas por ferrocarril entre las ciudades y sobre las tarifas correspondientes.

El vendedor desea conocer un circuito en tren que recorra cada ciudad exactamente una vez y regrese a la ciudad de partida, y cuya tarifa total sea mínima.

Problema del viajante

Representamos las soluciones como tuplas $(x_1, \ldots, x_n)$, donde x_i es el vértice por el que pasamos en i-ésimo lugar.

Problema del viajante

Representamos las soluciones como tuplas $(x_1, \ldots, x_n)$, donde x_i es el vértice por el que pasamos en i-ésimo lugar.

Las soluciones tienen que cumplir que se utilizan vértices válidos, que no hay repeticiones y que siempre hay arista de uno al siguiente, y que hay arista del último al primero.

Problema del viajante

Representamos las soluciones como tuplas $(x_1, \ldots, x_n)$, donde x_i es el vértice por el que pasamos en i-ésimo lugar.

Las soluciones tienen que cumplir que se utilizan vértices válidos, que no hay repeticiones y que siempre hay arista de uno al siguiente, y que hay arista del último al primero.

Para evitar soluciones repetidas fijamos el comienzo de los ciclos en el vértice 1, es decir, $x_1 = 1$, y empezaremos a tomar decisiones a partir del segundo vértice del ciclo.

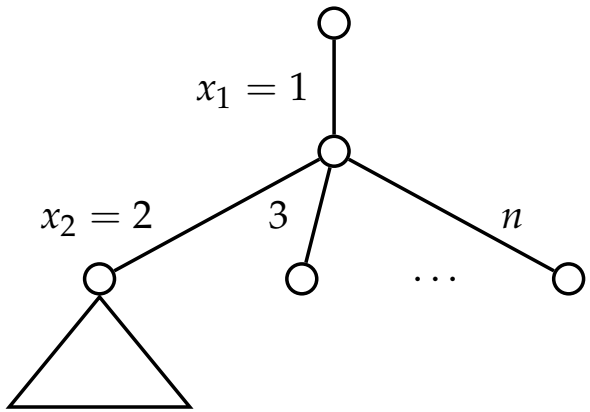
Problema del viajante

Representamos las soluciones como tuplas $(x_1, \ldots, x_n)$, donde x_i es el vértice por el que pasamos en i-ésimo lugar.

Las soluciones tienen que cumplir que se utilizan vértices válidos, que no hay repeticiones y que siempre hay arista de uno al siguiente, y que hay arista del último al primero.

Para evitar soluciones repetidas fijamos el comienzo de los ciclos en el vértice 1, es decir, $x_1 = 1$, y empezaremos a tomar decisiones a partir del segundo vértice del ciclo.

En el árbol de exploración cada nodo excepto la raíz tiene $n-1$ hijos, y hay n niveles.



Problema del viajante: cota

El coste de las soluciones será de la forma

$$\underbrace{\sum_{i=2}^{k} G.\texttt{valor}(x_{i-1}, x_i)}_{\text{fijo}} + \underbrace{\left(\sum_{i=k+1}^{n} G.\texttt{valor}(x_{i-1}, x_i) \right) + G.\texttt{valor}(x_n, x_1)}_{n-k+1 \text{ aristas}}$$

por lo que podemos aproximar el coste de las últimas $n-k+1$ aristas con $(n-k+1)\, mínG$, donde $mínG$ es el valor mínimo de todas las aristas de G.

Problema del viajante: cota

El coste de las soluciones será de la forma

$$\underbrace{\sum_{i=2}^{k} G.\texttt{valor}(x_{i-1}, x_i)}_{\text{fijo}} + \underbrace{\left(\sum_{i=k+1}^{n} G.\texttt{valor}(x_{i-1}, x_i) \right) + G.\texttt{valor}(x_n, x_1)}_{n-k+1 \text{ aristas}}$$

por lo que podemos aproximar el coste de las últimas $n-k+1$ aristas con $(n-k+1)\, mínG$, donde $mínG$ es el valor mínimo de todas las aristas de G.

Este problema puede no tener solución, lo que representamos con un coste igual a $+\infty$.

Problema del viajante: implementación

```
struct Nodo {
    vector<int> sol;
    int k;
    float coste;     // coste acumulado
    float coste_est; // prioridad
    vector<bool> usado;
};

bool operator<(Nodo const& a, Nodo const& b) {
    return a.coste_est < b.coste_est;
}

using Grafo = vector<vector<float>>;  // matriz de adyacencia

float calculo_minimo(Grafo const& g) {
    float minimo = numeric_limits<float>::infinity();
    for (int i = 0; i < g.size(); ++i) {
        for (int j = 0; j < g[i].size(); ++j) {
            minimo = min(minimo, g[i][j]);
        }
    }
    return minimo;
}
```

Problema del viajante: implementación

```
void vendedor_rp(Grafo const& grafo,
                 vector<int> & sol_mejor, float & coste_mejor) {
    int N = grafo.size();
    float minG = calculo_minimo(grafo);
    // se genera la raíz
    Nodo Y;
    Y.sol = vector<int>(N);
    Y.k = 0; Y.sol[Y.k] = 0;
    Y.usado = vector<bool>(N,false); Y.usado[0] = true;
    Y.coste = 0;
    Y.coste_est = N*minG;
    PriorityQueue<Nodo> cola;
    cola.push(Y);
    coste_mejor = INF;
    // búsqueda mientras pueda haber algo mejor
    while (!cola.empty() && cola.top().coste_est < coste_mejor) {
        Y = cola.top(); cola.pop();
        // generamos los hijos
        Nodo X(Y);
        ++X.k;
        int anterior = X.sol[X.k-1];
```

Problema del viajante: implementación

```
            for (int vertice = 1; vertice < N; ++vertice) {
                if (!X.usado[vertice] && grafo[anterior][vertice] != INF) {
                    X.sol[X.k] = vertice;
                    X.usado[vertice] = true;
                    X.coste = Y.coste + grafo[anterior][vertice];
                    if (X.k == N-1) {
                        if (grafo[vertice][0] != INF &&
                            X.coste + grafo[vertice][0] < coste_mejor) {
                            sol_mejor = X.sol;
                            coste_mejor = X.coste + grafo[vertice][0];
                        }
                    } else {
                        X.coste_est = X.coste + (N - X.k) * minG;
                        if (X.coste_est < coste_mejor) {
                            cola.push(X);
                        }
                    }
                    X.usado[vertice] = false;
                }
            }
        }
    }
```