ESQUEMA DE RAMIFICACION Y PODA: Branch & Bound

1. CARACTERIZACIÓN Y DEFINICIONES

2. EL ESQUEMA ALGORITMICO

3. LA EFICIENCIA

4. MOCHILA ENTERA

5. EL VIAJANTE DE COMERCIO

María Teresa Abad

Mayo 2005

1. CARACTERIZACION Y DEFINICIONES

El esquema de Ramificación y Poda es el algoritmo de búsqueda más eficiente en espacios de búsqueda que sean árboles.

En Ramificación y Poda los nodos del espacio de búsqueda se pueden etiquetar de tres formas distintas : nodo *vivo*, *muerto* o *en expansión*.

- Un nodo *vivo* es un nodo factible y *prometedor* del que no se han generado todos sus hijos.

- Un nodo *muert*o es un nodo del que no van a generarse más hijos por alguna de las tres razones siguientes :
  - ya se han generado todos sus hijos o
  - no es factible o
  - no es prometedor (un nodo es *prometedor* si la información que tenemos de ese nodo indica que expandiéndolo se puede conseguir una solución mejor que la mejor solución en curso).

En cualquier instante del algoritmo pueden existir muchos nodos vivos y muchos nodos muertos pero sólo existe un nodo *en expansión* que es aquél del que se están generando sus hijos en ese instante.

Vuelta Atrás es una búsqueda ciega mientras que Ramificación y Poda es una búsqueda *informada* $\Rightarrow$ la diferencia es el orden en que se recorren los nodos del espacio de búsqueda.

- *Recorrido ciego*: Fijado un nodo *x* del espacio de búsqueda el siguiente nodo a visitar es el primer hijo de *x* sin visitar, en un recorrido en profundidad y el siguiente hermano de *x*, en un recorrido en anchura.

- *Recorrido informado*: Fijado un nodo *x* del espacio de búsqueda el siguiente nodo es el más prometedor de entre todos los nodos vivos; es el que se va a convertir en el próximo nodo en expansión.

Otra diferencia importante:

- *Vuelta Atrás*: tan pronto como se genera un nuevo hijo del nodo en curso, este hijo se convierte en el nuevo nodo en curso o nodo en expansión y los únicos nodos vivos son los que se encuentran en el camino que va desde la raíz al actual nodo en expansión.

- *Ramificación y Poda*: se generan todos los hijos del nodo en expansión antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en expansión $\Rightarrow$ hay que conservar en algún lugar muchos más nodos vivos que pertenecen a distintos caminos. **Ramificación y Poda utiliza una estructura auxiliar para almacenar y manipular los nodos vivos.**

Distintos órdenes de recorrido del espacio de búsqueda:

1/ **estrategia FIFO** (la estructura auxiliar es una *cola*): da lugar a un recorrido del espacio de búsqueda por niveles o en anchura.

2/ **estrategia LIFO** (la estructura auxiliar es una *pila*) : produce un recorrido en profundidad aunque en un sentido distinto ('de derecha a izquierda' en lugar de 'de izquierda a derecha') al que se da en Vuelta Atrás.

3/ **estrategia LEAST COST** (la estructura auxiliar es una *cola de prioridad*): éste es el recorrido que produce Ramificación y Poda para un problema de minimización y usando función de estimación como la presentada en Vuelta Atrás.

> El valor de lo que un nodo vivo "promete" es la clave de ordenación (prioridad) en la cola de prioridad $\Rightarrow$
> **coste(x)** = *coste_real(x)+coste_estimado(x).*

2. EL ESQUEMA ALGORITMICO

- El nuevo nodo en expansión es el nodo vivo más prometedor de la lista de nodos vivos.

- Se generan todos sus hijos y a cada uno de ellos se les aplica el siguiente análisis :

  Sea *x* el hijo que se está analizando:
  - si *x* no es factible, entonces *x* pasa a ser un nodo muerto.

  - si *x* es factible y tiene un *coste(x)* peor que el de la mejor solución en curso, *x* pasa a ser un nodo muerto y nunca más se volverá a considerar.

  - si *x* es factible y tiene un *coste(x)* mejor que el de la mejor solución en curso pero *x* no es solución, entonces *x* se inserta con prioridad *coste(x)* en la cola de prioridad de nodos vivos.

  - si *x* es factible y tiene un *coste(x)* mejor que el de la mejor solución en curso y *x* es solución entonces pasa a ser la nueva mejor solución en curso. Además se revisan todos los nodos de la cola de prioridad de nodos vivos y se eliminan de ella todos aquellos que tengan una prioridad peor o igual que *coste(x)*.

- Una vez generados y analizados todos los hijos del nodo en expansión, éste se convierte en un nodo muerto y se vuelve a repetir el proceso.

- El proceso acaba cuando la cola de prioridad está vacía. En ese momento la mejor solución en curso se convierte en la solución óptima del problema.

Implementación

Hay que definir el contenido de un nodo $\Rightarrow$ debe almacenar toda la información que llegaba como parámetro a una llamada recursiva de Vuelta Atrás y también todo lo que era devuelto por ella.

Un *nodo* es una tupla que almacena:

- la solución en curso,
- la profundidad alcanzada en el espacio de búsqueda,
- el valor de todos los marcajes,
- el coste de la solución en curso y
- su coste estimado.

La *mejor solución en curso* será una variable de tipo nodo que a lo largo de las iteraciones del bucle se irá actualizando convenientemente.

```
función R&P (D es datos_probl ) dev ( x es nodo )
{ Pre : El espacio de búsqueda es un árbol y se trata de un problema de MINIMIZACION }
var
  x, y, xmejor es nodo;
  list es cola_prioridad(<nodo,valor>);
/* valor es la prioridad de la cola de prioridad */
  vmejor es valor;
fvar
      y := CREAR_RAIZ(D);
      list := INSERTAR(lista_vacia, <y, ESTIMAR(y) > )
      vmejor := algoritmo-externo(D);
      *[ ¬vacia(list) --->
            y := primero(list);    avanzar(list);
            /* y es el nodo vivo en expansión  */
            preparar_recorrido_hijos(y);
            *[ existan_hijos(y) --->
                  x:= siguiente_hijo(y);
                  x:= MARCAR(y, x);
                  est := ESTIMAR(x);
 /* est = coste_real(x)+coste_estimado(x) */
                  [ factible(x) ∧ prometedor(x,est)--->
                        [ solución(x) --->
                              <xmejor, vmejor>:=
                              <x, coste_real(x)>;
                              list:= DEPURAR(list,vmejor);
/* se han eliminado de la lista todos los nodos vivos que prometan una solución con un coste peor que el coste de la solución que se acaba de conseguir */
```

```
                    []¬solución(x) -->
                        list:= INSERTAR(list, <x, est >);
/* x es factible y prometedor, por tanto es un nodo vivo
que hay que insertar en la lista */
                    ]
                []¬factible(x) ∨ ¬ prometedor(x,est) --->
                        seguir;
                ]
            /* Desmarcar si es preciso */
        ] /* fin bucle existen_hijos */
    ] /* fin bucle lista_vacia */
```

{ ***Post****: Los nodos generados equivalen a un recorrido completo del espacio de búsqueda y en ese recorrido se ha encontrado el nodo que contiene la solución de mínimo coste real de entre todas las soluciones que hay para el problema. Ese nodo es* ***xmejor*** *y el valor de la solución es* ***vmejor****}*

```
    dev ( xmejor, vmejor )
ffunción
```

3. LA EFICIENCIA

El esquema de Ramifiación y Poda es la manera más eficiente que se conoce de recorrer un espacio de búsqueda que sea un árbol pero hay elementos que lo encarecen:

- Como hace un recorrido completo, en el peor de los casos, del espacio de búsqueda, el coste básico es proporcional al tamaño del EB (exponencial ¡!)

- Lo normal es usar PBCMSC. La efectividad de la PBCMSC depende de la calidad de la función de estimación. El coste de calcularla no ha de ser excesivo y es bueno disponer de una solución inicial para aumentar la eficiencia.

- La manipulación de la cola de prioridad de nodos vivos hace que el coste aumente. Las operaciones *primero*, *avanzar*, *insertar* y *depurar* son las que trabajan sobre la cola. De este modo *primero* tiene coste constante, *avanzar* e *insertar* tienen coste *log n*, y *depurar* tiene, en el peor de los casos, coste $n \cdot log n$, siendo n el tamaño de la cola de prioridad.

4. MOCHILA ENTERA

Definición de un nodo:

```
tipo nodo es tupla
    sol es vector[1..n] de {0,1};
                /* solución en curso*/
    k es nat;
                /* índice del objeto ≡profundidad*/
    pac es nat;
                /* peso acumulado */
    vac es nat;
                /* valor  acumulado ≡coste_real */
    est es nat;
                /* coste_estimado */
    ftupla
```

función MOCHILA_RP (xini es solución, vini es valor(xini)) dev (xmejor es nodo, vmejor es valor (xmejor))

*{ **Pre** : Los objetos a considerar están ordenados en orden decreciente de relación valor/peso ∧ **xini** es la solución inicial ∧ **vini** el valor de esa solución }*

```
    /* creación del nodo inicial */
    x.sol:= solución_vacía;
    x.k:= 0;
    x.pac:= 0;
```

```
    x.vac:= 0;
/* coste_estimado(x) :se resuelve mochila fraccionada
para una mochila capaz de soportar PMAX-pac y los
objetos del x.k+1 al n. Recordar que los objetos se
consideran en orden decreciente de relación valor/peso.
Devolverá la solución de mochila fraccionada para este
problema */
    x.est:= MOCH_FRAC( x, x.k+1, x.pac,PMAX );

/* creación de la cola de prioridad que contendrá los
nodos vivos en orden decreciente de clave porque el
problema es de maximización. La clave es x.vac+x.est
*/
    cp:= insertar(cola_vacia, <x, x.vac+x.est>);

/* inic. la mejor solución en curso */
    <xmejor,vmejor> := <xini,vini>;
    *[¬vacia(cp) --->
        y := primero(cp);    cp := avanzar(cp);
        i := -1;
        *[ i<1 ---> i := i+1 ;
                /* creación y llenado del hijo de y */
                x := y;
                x.k := y.k+1;
                x.sol[x.k] := i;
                x.pac := y.pac + (peso[x.k]*i);
                x.vac := y.vac + (valor[x.k] *i);
                x.est:=MOCH_FRAC(x,x.k+1,x.pac,PMAX );
```

```
                    [ (x.pac ≤ PMAX) ∧
                      (x.vac+x.est >vmejor) --->

/* la solución en curso es factible y prometedora */
                          [ x.k = n  --->
                          /* es una hoja y por tanto solución */
                                <xmejor,vmejor> := < x, x.vac>;
                                cp := depurar(cp, vmejor);
                          [] x.k < n  --->
                          /* no es una hoja, hay que seguir  */
                                cp:= insertar(cp,<x, x.vac+x.est>);
                          ]

                    [] (x.pac >PMAX) ∨
                       (x.vac+x.est ≤vmejor) ---> seguir;
                    ]

/* no hace falta desmarcar porque cada hijo se
inicializa siempre con el valor del padre */
               ]
        ]
{ Post : xmejor es la mejor solución en curso y
                    n
vmejor = Σx.sol[i]* valor[i]}
                  i=1
        dev ( xmejor, vmejor )
ffunción
```

La llamada a MOCHILA_RP va precedida por una llamada al algoritmo Voraz que calcula la solución para mochila fraccionada pero eliminando de esa solución el valor del objeto que se haya fraccionado. Basta con inicializar *vini* con el resultado del Voraz convenientemente modificado e inicializar *xini* a *nodo_vacio*.

5. EL VIAJANTE DE COMERCIO

Definición de un nodo:

```
tipo nodo es tupla
    sol es vector[1..n] de {1,n};
                    /* solución en curso*/
    k es nat;
                    /* profundidad*/
    ltour es nat;
                    /*longitud acumulada≡coste_real */
    lhnos es lista(vértices);
                    /* vértices del nivel k+1 */
    est es nat;
                    /* coste_estimado */
    ftupla
```

función TSP_RP (xini es solución, vini es valor(xini), C es matriz, n es nat) dev (xmejor es nodo, vmejor es valor (xmejor))

*{ **Pre** : **xini** es la solución inicial ∧ **vini** el valor de esa solución }*

```
    /* creación del nodo inicial */
    x.sol[1]:= 1;          /* partimos del nodo 1 */
    x.sol[2..n]:=0;
    x.k:= 2;
    x.ltour:= 0;
    x.lhnos:= <2,3,...,n>;
/* coste_estimado(x) :  */
    x.est:= F_ESTM( x, x.k);

/* creación de la cola de prioridad que devolverá los
nodos vivos en orden creciente de prioridad porque el
problema es de minimización.
La clave es x.ltour+x.est */
    cp:= insertar(cola_vacia, <x, x.ltour+x.est>);

/* inic. la mejor solución en curso */
    <xmejor,vmejor> := <xini,vini>;
    *[¬vacia(cp) --->
        y := primero(cp);    cp := avanzar(cp);
        i := 0;
        x:=y;
        *[ i < n-y.k --->
              i := i+1 ;
```

```
                /* llenado del hijo de y */
                x.k := y.k+1;
                x.sol[x.k] := primero(y.lhnos);
                y.lhnos:= avanzar(y.lhnos);
                x.lhnos := y.lhnos ;
/* podríamos mirar primero si existe la arista */
                x.ltour := y.ltour + (C[y.sol[y.k], x.sol[x.k]]);
                x.est:=F_ESTM(x,x.k);
                [ (x.est < vmejor) --->
/* la solución en curso es factible y prometedora */
                    [ x.k = n  --->
                    /* es una hoja y por tanto solución */
                        vmejor := x.ltour+C[x.sol[n],x.sol[1]];
                        xmejor := x;
                        cp := depurar(cp, vmejor);
                    [] x.k < n  --->
                    /* no es una hoja, hay que seguir  */
                        cp:= insertar(cp,<x, x.ltour+x.est>);
                    ]

                    [] (x.est ≥ vmejor ) ---> seguir;
                    ]
                    /* desmarcar */
                    y.lhnos := insertar ( y.lhnos, x.sol[x.k]);
            ]
    ]
{ Post : xmejor es un nodo que contiene un tour de coste
mínimo y vmejor es la longitud de ese tour }
dev ( xmejor, vmejor )
ffunción
```

Igual que hemos usado una función, F_ESTM, para calcular una cota inferior de la mejor solución, usaremos el algoritmo Voraz propuesto en Vuelta Atrás para calcular una solución inicial y efectuar la llamada a TSP_RP con los valores obtenidos copiados en xini y vini.