Una propuesta metodológica para la enseñanza de la programación dinámica

Ignacio Peláez, Francisco Almeida, Daniel González
Dtpo. Estadística, I. O. y Computación
Universidad de La Laguna
La Laguna
e-mail: ignaciopp@gmail.com,{falmeida, dgonmor}@ull.es

Resumen

Realizamos en este trabajo una revisión de aspectos metodológicos relacionados con la enseñanza/aprendizaje de la técnica algorítmica *Programación Dinámica*. Tras una reflexión sobre el material que encontramos en la bibliografía básica, observamos ciertas inconsistencias en los métodos y formas utilizados. Estos aspectos contribuyen a descontextualizar el método en relación con otras técnicas algorítmicas y a dificultar el proceso de aprendizaje. Introducimos una propuesta que considera aspectos metodológicos que deben ser tenidos en cuenta al presentar la técnica. La propuesta va acompañada de una herramienta de apoyo al profesor (Mallba-DP) para desarrollar algoritmos de Programación Dinámica en el marco de nuestra propuesta.

1. Introducción

La Programación Dinámica (PD) es una técnica algorítmica de tipo enumerativo que ha sido ampliamente estudiada en diversos campos como las Ciencias de la Computación, la Teoría del Control o la Investigación Operativa entre otros. Se trata de un método que habitualmente se enmarca entre las técnicas exactas de resolución de problemas y se asume que los alumnos de la titulación de Informática, así como los alumnos de otras titulaciones como Física, Matemáticas y muchas Ingenierías, deben tener conocimiento de ella. Es común, que en los currículos de informática aparezca entre los contenidos de asignaturas del tipo `Técnicas Algorítmicas`, `Estructuras de Datos y Algoritmos`, `Algorítmica`, etc. En este trabajo se pretende dar una visión de las dificultades que aparecen cuando se intenta introducir la técnica en las asignaturas mencionadas, haciendo especial énfasis en aquellos aspectos que hacen que la técnica, a menudo, desaparezca de los temarios o que, en ocasiones, quede relegada a aquellos capítulos estratégicamente situados al final del temario y que dejan de impartirse por falta de tiempo. En la literatura clásica, encontramos el inconveniente de que hay una cierta distancia entre los conceptos teóricos que plantea la técnica y, el algoritmo que finalmente que se obtiene de ella y que finalmente implementa el alumno. Este hecho lo observamos, por ejemplo, en la fórmula recursiva asociada a la PD, que suele aparecer como una fórmula mágica.

Se plantea en este trabajo una metodología que formula la técnica de un modo bastante intuitivo para el alumno, sin descuidar las consideraciones necesarias para introducir aquellos aspectos formales que permitan al alumno desarrollar algoritmos de forma metódica. Introducimos Mallba-DP, una herramienta de desarrollo de apoyo al profesor con la que ilustrar paso a paso la evolución de algorítmos de PD. Asimismo, como herramienta de desarrollo, es válida además para la implementación de algoritmos de PD por parte del alumno puesto que permite la monitorización de los algoritmos desarrollados.

Nos planteamos como objetivos: introducir de forma metodológica la PD de manera que el alumno pueda, de forma intuitiva, desarrollar nuevos algoritmos de Programación Dinámica del mismo modo que lo hace con otras técnicas; facilitar al alumno el desarrollo de programas de PD de acuerdo a la metodología propuesta, de modo que se reduzca la distancia entre la técnica formal y el programa desarrollado; por último, contribuir a mejorar el análisis de los resultados obtenidos al aplicar la PD la obtención de la política óptima.

Estructuramos el trabajo del siguiente modo: la sección 2 recoge una reflexión de aquellos aspectos que en nuestra opinión no están siendo tratados de forma adecuada en los libros de texto, la sección 3 introduce nuestra propuesta en dos pasos, en primer lugar se introduce un ejemplo sencillo y luego se plantea el método desde un punto de vista más general; la sección 4 presenta la herramienta Mallba-DP e ilustramos su uso en la sección 5. Finalizamos el trabajo

con una sección de conclusiones.

2. La Programación Dinámica en los libros de texto

Habitualmente la resolución de problemas mediante Programación Dinámica suele introducirse en las asignaturas después de que han sido presentadas al alumno otras técnicas de tipo enumerativo, como la técnica divide y vencerás, la vuelta atrás e incluso la ramificación y acotación. Este es, a menudo, un orden natural aceptado y que coincide con el orden que presentan libros de texto ampliamente utilizados en la bibliografía básica.

En el caso de estas técnicas enumerativas y, a excepción de la Programación Dinámica, la metodología seguida plantea la descomposición del problema original a resolver en subproblemas que se asocian a nodos de un árbol de búsqueda, en el que las aristas representan relaciones entre subproblemas. La solución al problema de partida viene como consecuencia de la exploración de este árbol de acuerdo con las características específicas de la técnica aplicada. Las fórmulas típicas utilizadas al introducir estos métodos suelen considerar un ejemplo con el que presentar la técnica y su posterior formalización mediante un algoritmo genérico o bien, se plantea en primer lugar el algoritmo general y se ilustra su uso a través de casos particulares. Estas técnicas mantienen como denominador común la notación utilizada y la representación del problema y de los subproblemas generados. Para todas ellas el alumno obtiene un método general con el que abordar la resolución de problemas y, en la práctica, el desarrollo teórico permite derivar algoritmos, y posteriormente programas, de un modo coherente.

En el caso de la Programación Dinámica, la situación difiere bastante de la anterior y encontramos aún ciertas inconsistencias que dificultan el proceso de enseñanza/aprendizaje. En primer lugar, la técnica introduce el principio de optimalidad "*...Una política óptima tiene la propiedad de que cualesquiera que sean el estado del sistema y la decisión que se tome en él, las decisiones resultantes son óptimas en los siguientes estados...* ", o en una forma más simple, "*...Una política óptima está compuesta de subpolíticas óptimas...*". En este principio se introduce una nueva terminología que difiere de la utilizada en técnicas anteriores y que generalmente resulta confusa al alumno. Conceptos como decisión o política no suelen ser definidos formalmente y, el propio principio de optimalidad tampoco suele resultar sencillo de asimilar por el alumno.

En segundo lugar, se introducen los conceptos de estados y etapas y, con frecuencia, no se encuentran asociados con los conceptos de descomposición del problema en subproblemas que han sido presentados en las técnicas previas. Se introducen también las ecuaciones funcionales de la Programación Dinámica, se trata de ecuaciones de recurrencia que permiten calcular los estados de la tabla de PD. Es cierto que una vez presentada al alumno la ecuación funcional (fórmula mágica), el cálculo de la tabla de PD no suele presentar dificultad. Sin embargo, una vez calculada la tabla de PD el alumno debe ser capaz por sí mismo de obtener la secuencia de decisiones que proporcionan la solución óptima. Este proceso puede resultar muy complicado sin la instrucción adecuada.

Además, el alumno no dispone de un mecanismo genérico con el que obtener la ecuación de recurrencia, lo que le impide ser capaz, por si mismo, de desarrollar algoritmos de Programación Dinámica para nuevos problemas. La ausencia de este mecanismo impide también la obtención de un esquema algorítmico general con el que abordar la técnica.

En ocasiones, aún cuando en el texto se hace un esfuerzo por integrar muchos aspectos de la técnica, los inconvenientes se derivan del ejemplo con el que se ilustra que no permite abstraer la técnica del caso particular.

Por último, un aspecto del todo olvidado en muchos libros de texto es el análisis de sensibilidad. Tal y como indica Smith [4] en su libro, en muchos problemas de optimización es deseable conocer la sensibilidad de la solución de un problema a los elementos que la componen. Este interés muchas veces se concreta en cuestiones del tipo *¿Qué ocurriría si...?*. Podemos estar interesados en averiguar cómo determinados cambios podrían afectar a la política óptima y, en tal caso, cómo se ve afectado su costo o beneficio. La tabla de Programación Dinámica proporciona un formidable punto de partida desde el que realizar este análisis, mostrando todas las soluciones a subproblemas del problema original. Sin embargo, es necesario disponer de algún mecanismo eficaz con el que obtener la secuencia de decisiones que compone la solución. Esta tabla puede crecer de forma considerable para entradas de tamaño no muy grande, su cálculo

manual es inviable.

Planteamos seguir una metodología que permita:

- Introducir la notación de la PD en el contexto de la notación seguida con el resto de técnicas algorítmicas, de modo que el alumno pueda generar por si mismo las ecuaciones funcionales.
- Introducir el principio de optimalidad de forma que el alumno sea capaz de asimilar su fundamento.
- Introducir un esquema algoritmico genérico aplicable a una gran variedad de problemas.
- Proporcionar un producto software de apoyo al profesor con el que obtener de forma automática la tabla de Programación Dinámica y las secuencias de decisiones que componen una solución a un subproblema. Así, el análisis de sensibilidad se integra como parte de la solución a un problema.
- Ilustrar la técnica con un amplio conjunto de ejemplos.

La base de la propuesta que hacemos la encontramos en el libro de Ibaraki [3]. No podemos considerarlo un libro para incluir en la bibliografía básica puesto que se trata de texto orientado a investigadores en el campo en cuestión. En él se introduce una formulación de la Programación Dinámica basada en la teoría de autómatas. Si bien se trata de una formulación genérica y muy elegante la formalización utilizada podría llegar a resultar excesiva para el nivel académico en el que pretende ser introducida la técnica. Sin embargo, es cierto que la presentación es ordenada y consistente en los aspectos que planteamos. Además introduce, clasifica y agrupa los problemas en clases de modo que se facilita la labor de diseño de algoritmos.

3. La Metodología

3.1. Un Ejemplo Sencillo

Tal y como sugieren Ibaraki [3] y Smith [4] proponemos plantear como primer ejemplo el problema de los caminos mínimos en un grafo multietapa (figura 1). Bajo nuestro punto de vista, el ejemplo presenta varias ventajas:

- Se trata de un problema de búsqueda en un grafo. El alumno ya está familiarizado con el concepto y la notación de grafo y, se mantiene en el contexto de las estructuras de datos (árboles) ya utilizadas en otras técnicas exactas.
- Las ecuaciones surgen de forma natural al calcular la distancia óptima al vértice destino desde los nodos (ciudades) a las que está conectado.
- El concepto de estado está claramente asociado con el de subproblema en el problema original.
- Las decisiones coinciden, por definición, con las aristas del grafo. Este hecho permite introducir el concepto de política óptima (solución óptima) como una secuencia de decisiones óptimas.
- El alumno descubre el principio de optimalidad cuando se le plantea la cuestión de la optimalidad en subcaminos del camino óptimo.
- El recorrido por etapas en el grafo permite además ilustrar cómo es descartada la consideración de soluciones parciales que con seguridad no darán lugar a soluciones óptimas.

Una buena reflexión sobre el análisis de sensibilidad para este problema la podemos encontrar en [4]. En este caso, el análisis nos permite observar cómo se ve afectada la política óptima por el incremento o decremento del coste de arcos del grafo que pueden pertenecer, o no, a dicha política.

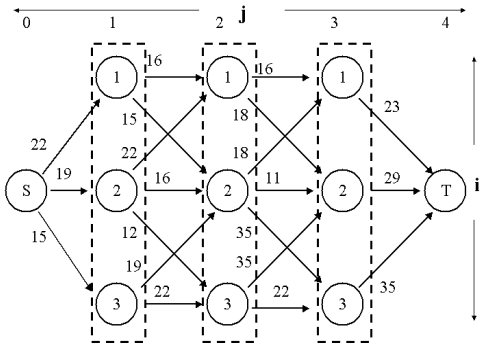

Figura 1: Grafo Multietapa

3.2. Generalizando la propuesta

Una vez que ha sido introducido un ejemplo con el que contextualizar la técnica se puede seguir la formalización que se plantea en [3] para derivar la técnica. Sin embargo, tal y como mencionamos en la sección 2, podemos relajar aquellos aspectos formales sin perder generalidad. Se puede seguir la estrategia que se plantea en el libro de Cormen y otros [2], en la que se indica que el desarrollo de un algoritmo de Programación Dinámica puede ser obtenido como una secuencia de cuatro pasos:

- Caracterizar la estructura de una solución óptima al problema.
- Definir de forma recursiva el valor de una solución óptima.
- Computar el valor de una solución óptima de modo $bottom - up$.
- Construir una solución óptima desde la información computada.

De la aplicación de esta secuencia de pasos podemos obtener para muchos problemas ecuaciones de recurrencia similares a las obtenidas para el problema de la sección 3.1. La dependencia entre estados en la tabla de Programación Dinámica deriva en un grafo multietapa y, para estos problemas basta con aplicar la técnica expuesta. Sin embargo en otras muchas ocasiones, la recurrencia obtenida es mucho más compleja. Planteado el método en estos términos, es el momento de introducir el conjunto de casos prácticos que permitan ilustrar grupos de problemas a ser abordados, de modo que ante una nueva situación el alumno tenga alguna referencia tipo.

Dos elementos que permiten agrupar problemas lo encontramos en el número de etapas implicadas simultáneamente en la recurrencia y, la cardinalidad de la dependencia que aparece en la fórmula, esto es, el número de términos recursivos simultáneos en la fórmula. Encontramos problemas multietapa cuando en la fórmula recurrente hay dependencia únicamente de etapas consecutivas como ocurre con el ejemplo de la sección 3.1. Obsérvese que la dependencia es entre las etapas n y $n-1$. Sin embargo, en la recurrencia $m[i,j] = min\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j; i \leq k < j\}$ que obtenemos en la parentización óptima de productos de matrices encadenadas [2], la dependencia no es únicamente entre etapas consecutivas (k varía entre i y j) por lo que el problema es no multietapa y, además la recurrencia es no monádica, el término recursivo aparece dos veces simultáneamente, $(m[i,k] + m[k+1,j])$. Dado un problema a resolver, el desarrollo de un algoritmo de Programación Dinámica se ve facilitado si se es capaz de identificar el grupo, en la clasificación de Programación Dinámica, al que pertenece. Para el caso multietapa, [1] encontramos un conjunto de suficientemente amplio de casos con los que ilustrar la técnica. El caso no monádico que aquí se presenta, con mucha frecuencia aparece descrito en los libros de texto y representa a una amplia variedad de problemas.

4. Mallba-DP

Un esqueleto Mallba es una pauta algorítmica que implementa un método genérico de resolución, y que ha de ser debidamente completado para instanciar y resolver un problema concreto. Los esqueletos algorítmicos proporcionan la base de funcionamiento genérica y el usuario completa sobre ellos una capa de definición con las características del problema que quiere resolver. La parte proporcionada por el esqueleto se implementa mediante clases en las que se indica explícitamente que son proporcionadas por el esqueleto. La parte que requiere ser completada por el usuario con la instanciación de su problema se implementa mediante clases en las que explícitamente se indica que son requeridas del usuario.

El esqueleto Mallba-DP, sigue el modelo de clases descrito por la librería Mallba y añade los elementos específicos relativas a la Programación Dinámica, manteniendo las características de eficiencia y facilidad de uso propias de la herramienta. Implementada en C++ abstrae los conceptos de Estado, Etapa y Decisión. El usuario debe describir el problema y la solución y, en particular, los métodos que permiten evaluar un estado y obtener la solución óptima al problema. Las clases proporcionadas por el esqueleto se encargarán de crear y computar la tabla de Programación Dinámica y de proporcionar aquellos métodos que permiten obtener la solución, ocultando los detalles de la implementación.

5. Un caso de estudio: El problema de la mochila 0/1

En esta sección haremos uso de la herramienta Mallba-DP en la resolución de un problema de optimización, el problema de la mochila 0/1. Se trata de un problema bien conocido que puede ser definido del siguiente modo: Dados un conjunto de N elementos y una mochila de capacidad C, se trata de seleccionar un subconjunto de los elementos de forma que se maximice el beneficio total $z = \sum_{j=1}^{n} p_j x_j$ sujeto a no exceder la capacidad de la mochila $\sum_{j=1}^{n} w_j x_j \leq C$, donde x_j es 1 si el elemento j es seleccionado y 0 en otro caso y, p_j y w_j denotan el peso y beneficio respectivamente para el elemento j.

Aplicada la metodología presentada en la sección 3, puede obtenerse para este problema su ecuación de recurrencia.

5.1. Las clases problema y solución

La clase problema representará al problema que se pretende resolver. Deberemos definir las estructuras para guardar la información referente a nuestro problema y especificar los métodos necesarios para manejar estas estructuras.

En nuestro ejemplo podemos considerar que un problema está definido mediante el número de objetos N, la capacidad de la mochila C y dos vectores w, p, que almacenarán los pesos y beneficios respectivamente. Debido a las restricciones de espacio, nos reflejaremos únicamente la estructura de datos (algoritmo 1).

```
requires class Problem {
   int N;        // Número de objetos
   int C;        // Capacidad
   int w[MAX_N]; // Pesos
   int p[MAX_N]; // Beneficios

   public:
   // Métodos para manipular la clase
      ....
};
```

Algoritmo 1: Definición de la clase Problem

La complejidad para definir un problema podrá variar en función del problema a estudiar, pero en la mayoría de los casos considerados, resulta bastante sencillo.

La clase solución por su parte será la encargada de representar una solución a nuestro problema.

Para nuestro ejemplo, una solución (ver algoritmo 2) nos debe proporcionar la información sobre cuál es el beneficio máximo así como cuales de los objetos deben ser introducidos en la mochila para conseguir ese beneficio. Definiremos un estado óptimo, que será el que nos proveerá el valor del beneficio máximo obtenido, también definiremos un vector de las decisiones tomadas a la hora de introducir objetos o no en la mochila. Para hacer más simple la definición de la clase solución usaremos dos clases nuevas, la clase estado y la clase decisión.

```
requires class Solution {
   State optimal_state;
   Decision *optimal_policy;

   public:
   // Métodos para manipular la clase
      ....
};
```

Algoritmo 2: Definición de la clase Solution

5.2. Las clases estado y decisión

La clase estado mantiene toda la información asociada a un estado de un algoritmo de programación dinámica. Esta clase almacena y calcula la información sobre el beneficio máximo en este estado y qué decisión se ha tomado para alcanzar este beneficio. En la evaluación de un estado puede ser necesario acceder a otros estados de la tabla de PD. Mallba-DP proporciona la clase `Table` que se gestionará de forma transparente para cada objeto de la clase `Solver`. Los métodos `GET_STATE(i, j)` y `PUT_STATE(i, j)` de la clase `Table` permiten obtener e insertar respectivamente estados en una tabla.

```
requires class State {
   const Problem &pbm;
   Solution &sol;
   int value;
   Decision d;
   Table &table;

   public:
```

```
   // Métodos para manipular la clase
      ....
   // Evalua un estado
   void Evalua(int stage, int index);

   // Devuelve el estado óptimo anterior
   void former_decision(State st,
    int current_stage, int current_index;
    int &former_stage, int &former_index);

};
```

Algoritmo 3: Definición de la clase `State`

El código 3 define la clase estado para nuestro ejemplo. Se definen: un problema mediante la variable `pbm`, una solución mediante la variable `sol`, una decisión mediante la variable `d` y la tabla que nos proporcionará acceso a todos los estados de nuestro problema mediante la variable `table`. Estas variables pueden considerarse genéricas, ya que deberían aparecer para cualquier problema que queramos resolver. Además se define la variable `value` que en este caso concreto almacenará el valor del beneficio máximo. Dos métodos merecen especial atención en esta clase, el método que evalúa un estado (`Evalua`) y el que permite transitar a un estado anterior en la tabla (`former_decision`). El primero de ellos implementa la evaluación del estado haciendo uso de la ecuación de recurrencia de la Programación Dinámica (ver algoritmo 4). El segundo, es necesario en el recorrido inverso de la tabla en busca de la solución.

```
void State::Evalua(int stage, int index){
  State st(pbm, sol, table);
  State tmp(pbm, sol, table);
   .....
  st = table.GET_STATE(stage-1, index);
  st.set_decision(0);
  if(index >= pbm.weight(stage)) {
    tmp = table.GET_STATE(stage-1,
                index-pbm.weight(stage));
    if (st.get_value() < (tmp.get_value()
                    + pbm.profit(stage)) {
      st.set_value(tmp.get_value() +
                   pbm.profit(stage));
      st.set_decision(1);
    }
  }
  table.PUT_STATE(st, stage, index);
   .....
}
 .....
```

Algoritmo 4: Implementación de la clase State

La clase `decision` permite representar la decisión tomada durante la evaluación de un estado. En nuestro ejemplo esta clase constará de un valor entero para almacenar un uno o un cero.

5.3. La función `main()`

Una vez definidas todas las clases necesarias, tan sólo queda hacer uso de estas para resolver el problema y mostrar los resultados deseados. Para esto lo que haremos será definir los pasos necesarios dentro de la función principal `main()`.

En el algoritmo 5 vemos un ejemplo de la implementación de la función `main`. Obsérvese que se incluye la librería KP.hh, esta librería contendrá las clases definidas anteriormente, así como las clases proporcionadas en el esqueleto de programación dinámica.

En nuestro ejemplo las variables `pbm`, `sol` y `setup` constituyen el problema, la solución a nuestro problema y la configuración de nuestro problema. Una vez los datos relativos al problema han sido cargados puede definirse el objeto `solver` de la clase `solver_seq`. Este objeto es el resolutor de nuestro problema. Para obtener la solución simplemente se invoca al método `run` del objeto `solver`, y automáticamente se evaluarán todos los estados de la tabla.

```
....
#include "KP.hh"
int main (int argc, char** argv) {
  Problem pbm;
  Solution sol(pbm);
  Setup setup;
  ifstream f1(argv[1]);
f1 >> setup;
ifstream f2(argv[2]);
{f2 >> pbm;
  Solver_seq solver(pbm, setup);
  solver.run();
  solver.show_table();
  solver.get_policy(
         setup.get_Num_Stages() - 1,
         setup.get_Num_States());
  sol = solver.solution();
```

```
  cout << sol << endl;
}
```

Algoritmo 5: Implementación de la función main

En este ejemplo una vez ejecutado el método `run` se muestra por pantalla la tabla obtenida. A continuación se busca la política óptima para este problema mediante la llamada al método `get_policy` que devuelve la política óptima a partir del estado final. El cálculo de ka polìtica se hace de forma transparente por la herramienta una vez que el usuario ha escrito el procedimiento `former_decision()`. Por último sólo nos queda mostrar por pantalla la información relativa a la solución.

5.4. Las salidas

A continuación veremos un ejemplo de la ejecución del programa para un problema de la mochila con entrada: $N = 5$, $C = 10$, $w = \{3, 11, 4, 6, 2\}$ y $p = 2, 20, 9, 3, 5$. La salida obtenida se ilustra en la figura 2. Se muestra la información relativa a los parametros usados en la ejecución: los parámetros de configuración (número de etapas y número de estados) y el problema (número de objetos, capacidad de la mochila, pesos y beneficios). Una vez calculada la solución se nos mostrará la tabla de programación dinámica. Esta tabla será la que finalmente nos proporciona el beneficio máximo y los objetos a insertar. La tabla nos muestra para cada estado el valor máximo y entre paréntesis la decisión tomada.

```
Stages: 5
States: 10

Objects: 5
Capacity: 10
Weights: 3 11 6 4 2
Profits: 2 20 9 3 5
 0(0)  0(0)  0(0)  2(1)  2(1)  2(1)  2(1)  2(1)  2(1)  2(1)  2(1)
 0(0)  0(0)  0(0)  2(0)  2(0)  2(0)  2(0)  2(0)  2(0)  2(0)  2(0)
 0(0)  0(0)  0(0)  2(0)  2(0)  2(0)  9(1)  9(1)  9(1)  11(1)  11(1)
 0(0)  0(0)  0(0)  2(0)  3(1)  3(1)  9(0)  9(0)  9(0)  11(0)  12(1)
 0(0)  0(0)  5(1)  5(1)  5(1)  7(1)  9(0)  9(0)  14(1)  14(1)  14(1)

Sulution: 14
Optimal Policy: 0 0 1 0 1
Time: 2.3e-05
```

Figura 2: Pantalla de ejecución

Después de mostrar la tabla, se imprimirá por pantalla la soluciòn, que será el benficio máximo obtenido, y la política óptima. En nuestro caso particular, la Figura 2 muestra como sólo se insertan en la mochila los objetos tres y cinco, que proporcionan un beneficio de catorce.

La tabla de programación dinámica nos muestra para cada estado cual es el valor de la evaluación del estado así como la decisión tomada al evaluar éste. Representaremos cada etapa como una fila y para cada etapa mostraremos el valor de cada estado en dicha etapa. El significado de cada estado depende del problema específico a estudiar y podemos generalizar que cada estado resuelve un subproblema del problema en estudio.

En nuestro ejemplo concreto, cada etapa representará un objeto a estudiar, por lo que tendremos tanta filas como objetos, y cada estado representará la capacidad de la mochila, comenzando desde cero hasta la capacidad especificada en nuestro problema. Cada estado nos dará la solución óptima para un subproblema de nuestro problema, así el elemento (i,j) de nuestra tabla nos dará la solución óptima al problema de elegir entre los i primeros objetos que deben ser introducidos en una mochila de capacidad j.

5.5. Análisis de sensibilidad

Mediante la tabla de programación dinámica es bastante sencillo hacer un estudio de sensibilidad, dado que la tabla no sólo contiene la soución al problema inicial sino también a subproblemas del problema inicial. Podemos suponer que la capacidad de la mochila puede variar, pero el aumento de la capacidad de la mochila nos supone un mayor costo, sería muy fácil calcular si el aumento de la capacidad supone un mayor beneficio. Obsérvese, que una mochila con capacidad igual a diez se proporciona un beneficio igual a catorce que es el mismo beneficio que se obtiene con capacidad igual a nueve y con capacidad igual a ocho, por lo que deberíamos seleccionar una mochila de capacidad ocho ya que con un menor costo obtenemos el mismo beneficio.

La función `main()` puede ser modificada por el usuario para especificar un rango mínimo y máximo donde podría variar la capacidad de la mochila y calcular el mejor benefico/costo, lo que facilita el análisis de sensibilidad.

6. Conclusiones

En este trabajo presentamos una revisión sobre cuestiones metodológicas en relación con la Progra-

mación Dinámica y mostramos aquellos aspectos que contribuyen a descontextualizar el proceso de enseñanza/aprendizaje. Presentamos un proceso metodológico que puede ser utilizado para presentar la asignatura de forma mejor integrada. La propuesta hace uso de la herramienta Mallba-DP que puede utilizar el profesor para ilustrar la técnica. Esta metodología será evaluada en la asignatura "Procesamiento informático de datos" de la licenciatura en Ciencias y Técnicas Estadísticas.

Referencias

[1] F. Almeida. Programación dinámica. www.pcg.ull.es.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.

[3] T. Ibaraki. *Enumerative Approaches to Combinatorial Optimization, Part II*. Annals of Operations Research. Volume 11, 1-4, 1988.

[4] David K. Smith. *Dynamic Programming. A practical Introduction*. Ellis Horwood, 1991.