Examen de Teoría de la Programación

E. U. ING. TEC. EN INFORMÁTICA DE OVIEDO

Final Septiembre – Curso 2005-2006 11 de septiembre de 2006

DNI _____ Nombre _____ Apellidos _____

Titulación: ☐ Gestión ☐ Sistemas

3. (1,5 puntos) El algoritmo de Floyd consiste en encontrar el camino más corto entre todos los pares de nodos de un grafo. Dado un grafo G, lo que queremos es hallar el camino más corto para ir desde un nodo *i* hasta otro nodo *j* de forma directa o a través de otro nodo *k*.
Para resolver este problema partimos de la matriz de pesos del grafo (D), es decir, una matriz que recoge el coste de ir de un nodo a otro del grafo sin pasar por nodos intermedios. Cuando no hay camino directo entre dos nodos, la celda correspondiente tendrá valor infinito. Y se aplica la siguiente función D(i,j)= min(D(i,j), D(i,k)+D(k,j)) pudiendo ser k cualquier otro nodo del grafo.

a) Razonar por qué decimos que Floyd se encuadra dentro de las técnicas de programación dinámica.

La solución depende de una función recursiva; pero en vez de darle una solución implementándola también recursivamente, representamos los datos en una tabla y vamos construyendo la solución sobre esta iterativamente, partiendo de los casos triviales (caminos directos) a los casos más complejos.

b) Representar gráficamente el array necesario para implementar el algoritmo, suponiendo que tenemos un grafo de 4 nodos.

Origen / Destino	1	2	3	4
1				
2				
3				
4				

c) Explicar que valores se pueden rellenar de forma directa en la tabla.

Los valores de los caminos directos.

d) Marcar sobre el array del apartado b) las casillas de las que depende (3,2) para calcular su valor.

Origen / Destino	1	2	3	4
1				
2				
3				
4				

No son sólo dos celdas ya que *k* varía entre 1 y 4

e) ¿Qué complejidad tiene este algoritmo?

O(n3)

(Problema expuesto en el grupo B)

4. (2 puntos) El problema del salto del caballo consiste en recorrer un tablero de ajedrez completo (8x8) a base de movimientos como los que realiza el caballo. Se quieren buscar todas las formas posibles de recorrer el tablero por la técnica de backtracking (vuelta atrás).
Las casillas a las que puede acceder un caballo desde una dada se muestran en la tabla adjunta.

	5		4	
6				3
		*		
7				2
	8		1	

Dada la clase *SaltoCaballo* que se muestra a continuación:

a) Escribir el método que realiza el backtracking, que llamaremos *ensayar*.

b) Escribir el método main() que hace la llamada a *ensayar*.

```
public class SaltoCaballo
{
	// Desplazamiento vertical del caballo
	private int[] despVertical=   {2, 1, -1, -2, -2, -1, 1, 2};
	// Desplazamiento horizontal del caballo
	private int[] despHorizontal= {1, 2, 2, 1, -1, -2, -2, -1};

	private int tam;                        // Tamaño del tablero
	private int[][] tablero;  // Estructura de datos para representar el estado

	/** Constructor */
	public SaltoCaballo(int iniTam)
	{	...	}

	/** Método que permite mostrar una solución */
	public void mostrarTablero()
	{	…	}

	/**
	 * Método que realiza el backtracking
	 * @arg mov, número de movimiento que se está realizando
	 * @arg x, coordenada x de la posición actual del caballo
	 * @arg y, coordenada y de la posición actual del caballo
	  */
	public void ensayar(int mov, int x, int y)
	{
		int a, b;
		// Bucle que prueba todos los posibles movimientos del caballo
		for (int k= 0; k<despVertical.length; k++)
		{
			a= x+despHorizontal[k];
			b= y+despVertical[k];
			// Comprobar que la posición es posible
			if (a>=0 && a<tam && b>=0 && b<tam && tablero[a][b]==0)
		/* Nueva posición debe estar dentro del tablero */
		/* La posición no está utilizada por un movimiento anterior */
			{
				tablero[a][b]= mov; // Anotar el movimiento en el tablero
				if (mov<tam*tam-1)
				// No hemos recorrido todas las casillas
				{
					// Seguir construyendo la solución
					ensayar(mov+1,a,b);
				}
				else
				{
					// Esta es una solución
					mostrarTablero();
				}
				tablero[a][b]= 0;   // borrar estado por la vuelta atrás
			}
		}
	}

	public static void main(String args[])
	{
		int tam= 8;  // Tamaño del tablero 8
		SaltoCaballo sc= new SaltoCaballo(tam);
		sc.ensayar(0,0,0);
	}
}
```

(Problema expuesto en el grupo C)

5. (1,5 puntos) Un turista desea realizar una ruta por 6 capitales europeas. El recorrido está establecido pero el turista podrá elegir cual es la ciudad origen, con lo cual queda determinado donde finalizará su recorrido. Cada ciudad tiene un coste, que serán los euros que cuesta pasar allí una noche. El turista puede recorrer a lo sumo 2 de estas ciudades en un día, con lo

cual deberá parar a dormir o bien en la primera que visite o visitar 2 y quedarse a dormir en la última.
El turista desea obtener el recorrido que cumpla las condiciones establecidas y que tenga coste mínimo. Disponemos de la clase Ciudad con los métodos públicos necesarios para acceder y modificar estos datos:

```
public class Ciudad {
  private String nombre;
  private int peso;
  private boolean visitado;
…
}
```

a) Razona por qué se debería utilizar un algoritmo devorador.

La principal característica de los algoritmos devoradores es que disponiendo de un heurístico son muy rápidos para buscar una solución óptima. Además, una vez encontrado un heurístico son fáciles de programar.

b) Describir un heurístico que proporcione una solución óptima.

Un heurístico fácil de implementar es "entre las dos ciudades a las que se puede mover el turista seleccionar la que tenga menor coste de hotel". Este heurístico es aceptable; aunque hay que remarcar que no siempre da la solución óptima. Un contraejemplo para demostrarlo:
Paris, 10
Londres, 30
Budapest, 40
Lisboa, 20
Madrid, 40
Roma, 10
Según nuestro heurístico pararíamos en: Paris, Londres, Lisboa, Roma con un coste de 70. Mientras que si paramos en Londres, Lisboa, Roma el coste es de 60.

c) Pseudocódigo del algoritmo devorador.

El pseudocódigo no haría falta detallarlo tanto; pero si utilizar la clase Ciudad proporcionada. Lo importante es: ver que estructuras de datos utilizamos, ver que existe un bucle y cuando finaliza y como se utiliza el heurístico para ir seleccionando ciudades.

```
public class TuristaEuropa
{
      int n= 6;
      Ciudad[] ciudades;
      int coste= 0;
      Ciudad[] solucion;

      // Constructor inicializará los arrays con el espacio suficiente

      public void devorador(int n)
      {
            int i= 0; int j= 0;
            while (!ciudades[i].getvisitado() && i<n)
            {
                  if (ciudades[i].peso < ciudades[i+1].peso)
                  {
                        ciudades[i].setVisitado(true);
                        solucion[j]= ciudad[i];
                        coste+= ciudades[i].peso;
                        i++;
                  }
                  else
                  {
                        ciudades[i+1].setVisitado(true);
                        solucion[j]= ciudad[i+1];
                        coste+= ciudades[i+1].peso;
                        i++;
                  }
```

```
                    j++;
                }
            }
        }
}
```

(Problema expuesto en el grupo A)

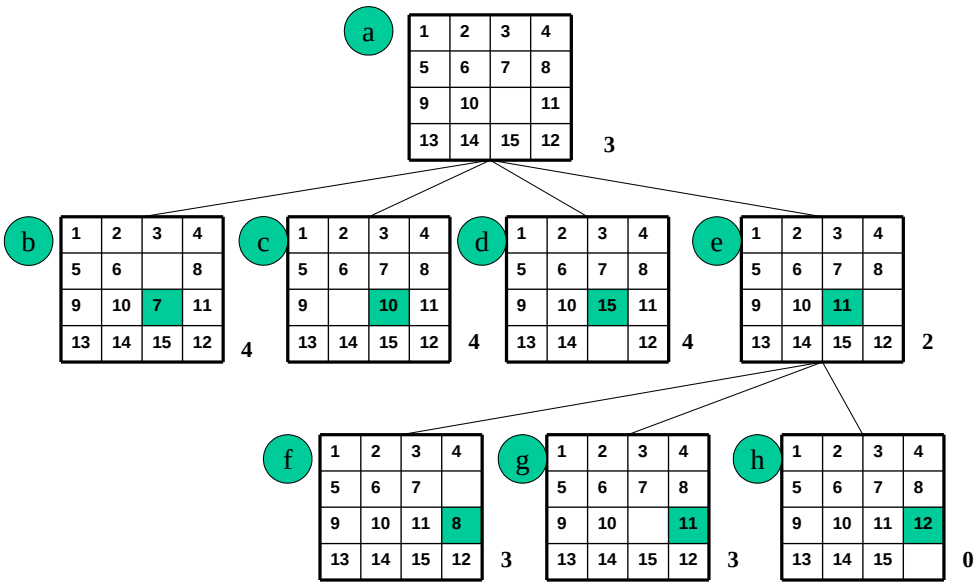
6. (2 puntos) El problema del puzzle se desarrolla sobre un tablero de 16 posiciones, donde hay colocadas 15 fichas, quedando una posición vacía. Las fichas no se pueden levantar del tablero, por lo que sólo es posible su movimiento por medio de desplazamientos sobre el mismo.

Tabla 1. Estado final

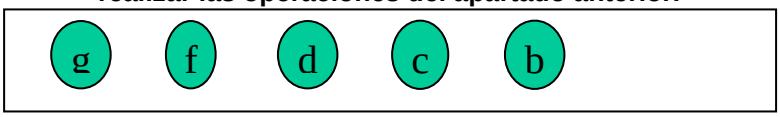
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

El objetivo del juego es, a partir de un estado inicial dado, obtener un estado objetivo (el mostrado arriba, tabla 1). Los únicos movimientos válidos son aquellos en que una ficha adyacente a la posición libre, se mueve hacia ésta. Se pueden ver estos movimientos permitidos como un desplazamiento de la casilla vacía hacia una de sus cuatro vecinas.
Queremos resolver este problema mediante la técnica de ramificación y poda. Y utilizamos el siguiente heurístico de ramificación: Calcular el número de piezas que están en una posición distinta de la que les corresponde en la disposición final.
Se pide:

a) **Representa gráficamente el árbol de estados de la técnica ramifica y acota que partiendo del estado que damos a continuación (tabla 2) desarrolla dos de sus nodos.**



b) **Representa gráficamente los estados que quedan en la cola de prioridad después de realizar las operaciones del apartado anterior.**



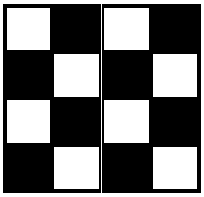
c) Explica las ventajas para este problema en concreto, de haber empleado esta técnica en vez de backtracking.

Este problema tiene un árbol que tiene mucha profundidad; pero es pequeño en anchura; por tanto el explorar el árbol en anchura no supone un sobre-coste grande; sin embargo la información que nos puede dar el heurístico puede significar que nos encaminemos rápidamente a la solución y sólo tengamos que explorar una pequeña parte del árbol. Además, en este problema tenemos el problema añadido de los estados repetidos que pueden llevar a ciclos sin fin en backtracking.

d) Justifica si hay alguna posibilidad de desarrollar algún heurístico de poda. Si es así proporciona uno y comenta brevemente como se implementaría.

A partir de lo visto en el problema incluido en el cuaderno didáctico, se pueden plantear dos heurísticos de poda:
a) determinar si a partir de un cierto estado, se puede alcanzar un estado objetivo o no.

- Numerando las casillas de 1 a 16 definimos:
  - posición(i): posición en el estado inicial de la ficha número i. posición(16) será la posición en el tablero de la casilla vacía.
  - menor(i): es el número de fichas j tales que j<i y posición(j)>posicion(i)
- El estado objetivo será alcanzable desde un estado si y sólo si *Sumatorio(menor(i)+x)* es par, donde x es 1 si la casilla vacía se encuentra en una de las casillas sombreadas del tablero adjunto y 0 si no es así.



$$\sum_{i=1}^{16} menor(i) + x$$

b) Eliminar estados repetidos ya explorados. Ir guardando en una cola los estados desarrollados para poder compararlos con los nuevos estados si aparece uno repetido automáticamente se podaría para evitar hacer la misma exploración por dos caminos.

Tabla 2. Estado inicial para el problema

1	**2**	**3**	**4**
5	**6**	**7**	**8**
9	**10**		**11**
13	**14**	**15**	**12**