UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA	**Programación III** Solución	**Prueba Presencial** **Primera Semana** Enero - Febrero de 2006 Duración: **2 horas** Material permitido: **NINGUNO**

Cuestión 1 (2.5 puntos). ¿En qué se diferencia una búsqueda ciega en profundidad y un esquema de vuelta atrás? (0.5 puntos)

La búsqueda ciega explora todas las ramas alternativas mientras que en un esquema de vuelta atrás se establecen condiciones de poda que determinan si una rama puede alcanzar o no una solución final. Si se determina que no es posible, entonces no se prosigue la búsqueda por dicha rama (se poda). Los problemas aptos para un esquema de vuelta atrás permiten expresar los nodos intermedios como soluciones parciales al problema. Aquellos nodos que no sean soluciones parciales no permiten alcanzar una solución final y, por tanto, su rama se poda. Por ejemplo, el problema de "poner N reinas en un tablero de NxN sin que se amenacen entre sí", requiere ir solucionando la siguiente secuencia de soluciones parciales:

poner 1 reina en un tablero de NxN sin que esté amenazada (trivial)
poner 2 reinas en un tablero de NxN sin que se amenacen entre sí
...
poner k reinas en un tablero de NxN sin que se amenacen entre sí
...
poner N reinas en un tablero de NxN sin que se amenacen entre sí (solución final)

Si por una rama no se puede resolver el problema para k, entonces evidentemente no se podrá resolver para N, por muchos intentos de añadir reinas hagamos.

En el espacio de búsqueda, ¿qué significa que un nodo sea k-prometedor? (1 punto)

Significa que es solución parcial para las k primeras componentes de la solución y, por tanto, todavía es posible encontrar una solución final (incluyendo las N componentes). En el problema de las N reinas habríamos colocado k reinas sin que se coman entre sí.

¿Que hay que hacer para decidir si un vector es k-prometedor sabiendo que es una extensión de un vector (k-1)-prometedor? (1 punto)

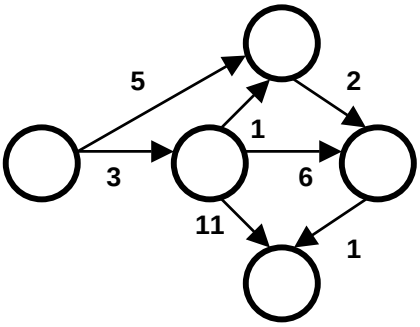
Si es (k-1)-prometedor quiere decir que es solución parcial para las primeras k-1 componentes y que, por tanto, estas cumplen las restricciones necesarias entre sí y no hay que volver a verificarlas. Entonces, para decidir si una extensión considerando la siguiente componente k conforma un nodo k-prometedor, lo único que hay que hacer es verificar si esta nueva componente k cumple las restricciones respecto a las otras k-1.

Cuestión 2 (1.5 puntos). Declara en Java o en Módula-2 las clases y/o estructuras de datos que utilizarías en el problema del Sudoku (práctica de este año) para comprobar en un tiempo de ejecución constante respecto a la dimensión n del tablero (nxn) que una casilla contiene un valor factible.

Existen varias alternativas, pero de nada sirve que verificar una casilla se realice en tiempo constante si luego actualizar su valor se realiza en tiempo lineal. En la siguiente solución se declaran tres tablas de booleanos que indican si ya hay o no un determinado valor en una determinada fila, columna o región, respectivamente. Si no es así, entonces es factible poner el valor en la casilla. Tanto la función de verificación como las de actualización tienen un coste computacional constante.

```
public class Tablero {

	int N=9;
	int subN=(int)Math.sqrt(N);

	boolean val_en_fil[][] = new boolean[N][N];
	boolean val_en_col[][] = new boolean[N][N];
	boolean val_en_reg[][] = new boolean[N][N];

	boolean valorFactible(int fil, int col, int val){
		int reg=region(fil,col);
		return (!val_en_fil[fil][val] &&
				!val_en_col[col][val] &&
				!val_en_reg[reg][val]);
	}

	void poner(int fil, int col, int val) {
		int reg=region(fil,col);
		val_en_fil[fil][val]=true;
		val_en_col[col][val]=true;
		val_en_reg[reg][val]=true;
	}

	void quitar(int fil, int col, int val) {
		int reg=region(fil,col);
		val_en_fil[fil][val]=false;
		val_en_col[col][val]=false;
		val_en_reg[reg][val]=false;
	}

	int region(int fil, int col) {
		return (col/subN)*subN+fil/subN; // división entera
	}
}
```

Cuestión 3 (2 puntos). Dado el siguiente grafo, rellena la tabla adjunta indicando paso a paso cómo el algoritmo de Dijkstra encuentra todos los caminos de menor coste desde el nodo 1.



Paso	Nodos Seleccionados	Nodos No Seleccionados	Vector de distancias				Vector de predecesores			
			2	3	4	5	2	3	4	5
0	{1}	{2,3,4,5}	3	∞	5	∞	1	1	1	1
1	{1,2}	{3,4,5}	3	9	4	14	1	2	2	2
2	{1,2,4}	{3,4}	3	6	4	14	1	4	2	2
3	{1,2,3,4}	{5}	3	6	4	7	1	4	2	3

Problema (4 puntos). Una empresa de montajes tiene n montadores con distintos rendimientos según el tipo de trabajo. Se trata de asignar los próximos n encargos, uno a cada montador, minimizando el coste total de todos los montajes. Para ello se conoce de antemano la tabla de costes C[1..n,1..n] en la que el valor c_{ij} corresponde al coste de que el montador i realice el montaje j. Se pide:

1. Determinar qué esquema algorítmico es el más apropiado para resolver el problema. Razonar la respuesta. (0.5 puntos)

   Se trata de un problema de optimización con restricciones. Por tanto, podría ser un esquema voraz o un esquema de ramificación y poda. Sin embargo descartamos el esquema voraz porque no es posible encontrar una función de selección y de factibilidad tales que una vez aceptado un candidato se garantice que se va alcanzar la solución óptima.

   Se trata, por tanto, de un algoritmo de ramificación y poda.

2. Escribir el esquema general (0.5 puntos)

```
Función RamificaciónPoda (nodo_raíz) dev nodo
    Montículo:=montículoVacío();
    cota:=acotar(nodo_raíz);
    poner((cota,nodo_raíz), Montículo);
    mientras no vacío(Montículo) hacer
        (cota, nodo):=quitarPrimero(Montículo);
        si solución(nodo) entonces
            devolver nodo;
        si no
            para cada hijo en compleciones(nodo) hacer
                cota:=acotar(hijo);
                poner((cota,hijo),Montículo);
            fpara;
        fsi;
    fmientras
    devolver Ø
```

3. Indicar que estructuras de datos son necesarias (0.5 puntos)

   nodo=tupla
   - asignaciones: vector[1..N];
   - último_asignado: cardinal;
   - filas_no_asignadas: lista de cardinal;
   - coste: cardinal;

   Montículo de mínimos (cota mejor la de menor coste)

4. Desarrollar el algoritmo completo (2.5 puntos)

Las funciones generales del esquema general que hay que instanciar son:

a. solución(nodo): si se han realizado N asignaciones (último_asignado==N)
b. acotar(nodo,costes): nodo.coste + “mínimo coste de las columnas no asignadas”
c. compleciones(nodo,costes): posibilidades para la siguiente asignación (valores posibles para asignaciones[último_asignado+1])

```
Función asignación(costes[1..N,1..N]) dev solución[1..N]
    Montículo:=montículoVacío();
    nodo.último_asignado=0;
    nodo.coste=0;
    cota:=acotar(nodo,costes);
    poner((cota,nodo),Montículo);
    mientras no vacío(Montículo) hacer
        (cota,nodo):=quitarPrimero(Montículo);
        si nodo.último_asignado==N entonces
            devolver nodo.asignaciones;
        si no para cada hijo en compleciones(nodo,costes) hacer
            cota:=acotar(hijo,costes);
            poner((cota,hijo),Montículo);
        fsi;
    fmientras
    devolver ∅;

Función acotar(nodo,costes[1..N,1..N]) dev cota
    cota:=nodo.coste;
    para columna desde nodo.último_asignado+1 hasta N hacer
        mínimo=∞;
        para cada fila en nodo.filas_no_asignadas hacer
            si costes[columna,fila]<mínimo entonces
                mínimo:=costes[columna,fila];
            fsi
        fpara
        cota:=cota+mínimo;
    fpara
    devolver cota;

Función compleciones(nodo,costes[1..N,1..N]) dev lista_nodos
    lista:=crearLista();
    para cada fila en nodo.filas_no_asignadas hacer
        hijo:=crearNodo();
        hijo.último_asignado:=nodo.último_asignado+1;
        hijo.asignaciones=nodo.asignaciones;
        hijo.asignaciones[hijo.último_asignado]:=fila;
        hijo.coste:=nodo.coste+costes[hijo.último_asignado,fila];
        hijo.filas_no_asignadas:=nodo.filas_no_asignadas;
        eliminar(fila,hijo.filas_no_asignadas);
        añadir(hijo,lista);
    fpara;
    devolver lista;
```