Soluciones - práctico 7

Algoritmos con retroceso (Backtracking)

EJERCICIO 1 (R)

Una casa de música ofrece como servicio un sitio web en el cual permite la descarga de ringtones. La casa de música ofrece *n* ringtones diferentes, donde de cada uno de estos se conoce: su precio, su ranking y su tamaño de descarga en bytes.

Un adolescente quiere descargar un conjunto de ringtones tal que, el tamaño de descarga total sea el más bajo posible, el precio total de dicha elección no supere su presupuesto P, y a su vez, el total del valor del ranking de los ringtones seleccionados, sea como mínimo R. Téngase en cuenta que no se pueden repetir ringtones.

Asuma que se tienen definidas las siguientes funciones:

1) *p(k)* indica el precio del *k*-ésimo ringtone con $1 \leq k \leq n$.
2) *r(k)* indica el ranking del *k*-ésimo ringtone con $1 \leq k \leq n$.
3) *d(k)* indica el tamaño de descarga en bytes del *k*-ésimo ringtone con $1 \leq k \leq n$.

Se pide:

a) Formalizar el problema en términos de Backtracking. Indicar: forma de la solución, restricciones explícitas e implícitas, función objetivo y predicados de poda.
b) Implementar la solución utilizando la estrategia BackTracking, indicando las porciones de código que se corresponden con las diferentes partes de la formalización.

Solución:

<u>Formalización:</u>

Forma de solución:

Tupla de largo fijo *n*, de la forma $< x_0, ,,, , x_{n-1} >$ donde *n* es la cantidad de ringtones.

Restricciones Explícitas:

$x_i \in \{0,1\}$ para todo *i* en {0, ,, , n-1}.

Restricciones Implícitas:

El precio total no debe superar el presupuesto *P*

$$\sum_{i=0}^{n-1} p[\mathrm{i}+1]\,\mathrm{x}_\mathrm{i} \leq P$$

El total del valor del ranking de los ringtones sea como mínimo *R*

$$\sum_{i=0}^{n-1} r[\mathrm{i}+1]\,\mathrm{x_i} \geq R$$

Función Objetivo:

El tamaño de descarga total sea lo más bajo posible, o sea

$f = \min_{t \in T} \{tam(t)\}$ donde $\mathrm{T} = \{\mathrm{t} = <\mathrm{x}_0, \ldots, \mathrm{x_n}> / \text{ t es solución}\}$ siendo $\mathrm{tam(t)} = \sum_{i=0}^{n-1} d[\mathrm{i}+1]\,\mathrm{x_i}$.

Predicado de Poda:

Se poda si la suma de los rankings de la tupla parcial en construcción $<\mathrm{x}_0, \ldots, \mathrm{x_k}, ?, ?, \ldots, ?>$ más la suma de los rankings de los ringtones que aún no han sido considerados no supera a *R*., o sea

$$\sum_{i=0}^{k} r[\mathrm{i}+1]\,\mathrm{x_i} + \sum_{i=k+1}^{n-1} r[\mathrm{i}+1] < R .$$

Implementación:

```
struct tupla
{
	int * ringtones;
	int precio;
	int ranking;
	int tam;
}

void search (tupla & t, int rankingRestantes, int pos, tupla &sol, int * p, int * r, int * d)
{
	if (t.precio <= P) // Restricción implícita (El precio total no debe superar el presupuesto P)
	{
		// Rest. implícita en caso 1 y Predicado poda en caso 2 (**)
		if (t.ranking + rankingRestantes >= R)
		{
			if (t.tam <= sol.tam) //Poda de la fun. objetivo
			{
				// caso 1: tupla solución
				If (pos == n) // Tupla Solución
				{
					for(int j=0; j<n;j++)
					{
						sol.ringtones[j] = t.ringtones[j];
					}
					sol.precio = t.precio;
					sol.ranking = t.ranking;
					sol.tam = t.tam;
				}
				else
				{
					// caso 2: sigue construyendo tupla
					for (int i=0; i<=1; i++) //Restricción explícita
					{
						rankingRestantes = rankingRestantes - r[pos];
						t.ringtones[pos] = i;
```

```
                                        t.precio = t.precio + i*p[pos];
                                        t.ranking = t.ranking + i*r[pos];
                                        t.tam = t.tam + i*d[pos];

                                        search(t,rankingRestantes, pos+1,sol, p,r,d);

                                        rankingRestantes = rankingRestantes + r[pos];
                                        t.precio = t.precio - i* p[pos];
                                        t.ranking = t.ranking - i* r[pos];
                                        t.tam = t.tam – i*d[pos];
                                   }
                              }
                         }
                    }
               }
}
```

Comentario:

(**)

En el caso 1: Es restricción implícita (la suma de rankings de sea por lo menos R) porque por como se actualiza el valor de la variable rankingRestantes, cuando se llegue a controlar si se alcanzó una solución "if (pos == n)", esta variable tiene como valor 0 y por lo tanto se controla en este if que t.Ranking sea >= R. Por lo tanto, no es necesario agregar la pregunta "if (t.ranking >= R)" Restricción implícita dentro del "if (pos == n)".

En el caso 2: Es predicado de poda porque controla que la suma de los ranking de los elementos de la tupla más la suma de los ranking de los elementos no considerados aun para formar parte de la tupla, permitan alcanzar por lo menos el valor de ranking R.

Invocación:

```
//los arreglos p,r y d se asumen ya cargados previamente
tupla t, sol;
t.ringtones = new int[n]; //no se inicializa el vector porque van tomando valores en el algoritmo de BK
t.precio = 0;
t.ranking = 0;
t.tam =0;
sol.ringtones = new int[n]; //no se inicializa el vector porque van tomando valores en el algoritmo de BK
sol.precio = 0;
sol.ranking = MAX_INT;
sol.tam = 0;
int rankingRestantes=0;

for (int i=0;i<n;i++) {
          rankingRestantes = rankingRestantes + r[i];
}
search(t, rankingRestantes, 0, sol, p, r, d);
// procesar sol
.
.
.
delete [] t.ringtones;
delete [] sol.ringtones;
```

Ejercicio 7 (R)

Escriba un programa en C* que dado un laberinto representado por una matriz de tamaño *N***M* de caracteres, encuentre el camino de salida del mismo. El algoritmo recibirá coordenadas de comienzo e imprimirá el camino hasta la salida.

Los mapas cuentan con 3 tipos de elementos:

'#' - Pared.
' ' - Camino.
'**S**' - Salida.

Tal como puede verse a continuación, para *N*=7 y *M*=9:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	#	#	#	#	#	#	#	#	#
1	#								**S**
2	#		#		#	#	#	#	#
3	#		#	#	#				#
4	#		#				#		#
5	#				#	#	#		#
6	#	#	#	#	#	#	#	#	#

Partiendo del nodo (3,1) el camino de salida del laberinto es:

(3,1), (2,1), (1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (1,7), (1,8).

Solución:

Se llamara 'L' a la matriz laberinto dada. Donde L[x][y] $\in$ {' ', 'S', '#'}, $0 \leq x < N$, $0 \leq y < M$.

Formalización:

Forma de la solución:

- Tupla de largo variable ***n+1*** de la forma $<w_0, w_1, \ldots, w_n>$, donde w_i corresponde a una coordenada (x_i, y_i) del mapa dado.

Restricciones explícitas:

En principio los valores de cada elemento de la tupla podrian ser coordenadas (x, y) tales que L[x][y] $\in$ {' ', 'S'} o sea:

$$w_i = (x_i, y_i)\ donde\ elemento[x_i][y_i] \in \{' ', 'S'\}, 0 \leq x_i, y_i < n\ \forall i, 0 \leq i \leq n$$

El planteo anterior no refleja las restricciones explicitas del problema. De acuerdo al problema la coordenada inicial es dada, la final deberia tener el valor 'S' unicamente y los valores intermedios deberian ser ' '.
De acuerdo a lo anterior las restricciones explícitas son:

4) $w_i = (x_i, y_i) \Rightarrow 0 \leq x_i < N,\ 0 \leq y_i < M,\ \forall i\ 0 \leq i \leq n$
5) $w_0 = (x_0, y_0)$ corresponde a la coordenada de comienzo *(es explícita por ser dato).*
6) $w_n = (x_n, y_n)$ ***donde*** $L[x_n][y_n]$ **= 'S'** *(la salida debe tener el valor 'S').*
7) $\forall\ i\ 0 \leq i < n,\ w_i = (xi, y_i)\ /\ L[x_n][y_n]$ **= ' '**

Restricciones implícitas:

Para cada coordenada $\boldsymbol{w_i}$ ***y*** $\boldsymbol{w_{i+1}}$ de la tupla solución, se cumple que $\boldsymbol{w_i}$ es adyacente a $\boldsymbol{w_{i+1}}$, con $\boldsymbol{0 \leq i < n}$.
Se dice que $\boldsymbol{w_i}$ **y** $\boldsymbol{w_{i+1}}$ son adyacentes $\Leftrightarrow$ $\boldsymbol{w_i}$ es ***(x, y)*** entonces $\boldsymbol{w_{i+1}}$ es ***(x, y+1) ó (x+1, y) ó (x-1, y) ó (x, y-1)***.

$$\forall w_i = (x_i, y_i), 0 \leq i < n, w_{i+1} = \begin{cases} (x_i, y_i + 1) \\ (x_i + 1, y_i) \\ (x_i - 1, y_i) \\ (x_i, y_i - 1) \end{cases}$$

Predicados de poda:

No hay.

Implementación:

Se utilizan los siguientes tipos auxiliares:

```
// Tipo Coord, modela coordenadas.
struct Coord
{
     int x, y;
};

const int N = 5, M = 5;
```

```
struct Stack;

// Constructoras

Stack* crearStack ();
// Crea el stack vacio

void agregarStack (Stack* &st, Coord c);
//Agrega la coordenada c al stack.

// Predicado

bool esVacioStack (Stack* st);
// Retorna true sii el stack es vacio.

// Selectoras

Coord* primeroStack (Stack* st);
// Retorna el primer elemento del Stack.
// Pre: !esVacioStack (st)

Stack* restoStack (Stack* st);
// Retorna el stack sin su primer coordenada.
// Pre: !esVacioStack (st)

// Destructora

void destruirStack (Stack* &st);
```

La implementacion del algoritmo en lo relativo a lo movimientos posibles especificados en la restriccion implicita:

$$\forall w_i = (x_i, y_i), 0 \leq i < n, w_{i+1} = \begin{cases} (x_i, y_i + 1) \\ (x_i + 1, y_i) \\ (x_i - 1, y_i) \\ (x_i, y_i - 1) \end{cases}$$

Que puede verse como wi+1 = (xi, yi) + $\left\{ \boxed{\begin{array}{l} (0, 1) \\ (1, 0) \\ (-1, 0) \\ (0, -1) \end{array}} \right.$ donde el recuadro se implementa con un stack (movs)

El algoritmo es:

```
/*
 * Algoritmo que resuelve el problema del Laberinto
 * Esta solución imprime todos los caminos desde el nodo
 * actual hasta la salida indicada por el carácter 'S'
 *
 * mapa       - corresponde a la matriz L
 * actual     - coordenadas de posición actual
 * visitado   - matriz usada para marcar los nodos visitados
 * solParcial - Stack camino solución hasta el momento
 */
void solucionLaberinto (char** mapa, Coord actual, int** visitado,
     Stack* solParcial, Stack* movs)
{
     if ((actual.x >= 0) && (actual.x < N) &&      //fun. de poda, 1era restr. explicita
            (actual.y >= 0) && (actual.y < M) &&   //fun. de poda, 1era restr. explicita
            (mapa[actual.x][actual.y] != '#') &&
            (!visitado[actual.x][actual.y]))
     {
            visitado[actual.x][actual.y] = 1;
            agregarStack (solParcial, actual);

            if (mapa[actual.x][actual.y] == 'S')   //fun. de poda, 3era restr. explicita
            {
                   // Encontré la salida, imprimo el camino solución.
                   imprimirStack (solParcial);
                   printf ("\n");
            }
            else
            {      // 4ta restr. explicita: en este bloque el elemento debe ser un ' ' y es
                   //  posible moverse

                   // Intento avanzar en las direcciones Norte, Sur, Este, Oeste
                   Stack* t_movs = movs;
                   while (!esVacioStack (t_movs))
                   {
                          Coord* c = primeroStack (t_movs);
                          t_movs = restoStack (t_movs);
                          Coord nuevo = actual;
                          nuevo.x += c->x;
                          nuevo.y += c->y;
                          solucionLaberinto (mapa, nuevo, visitado, solParcial, movs);
                   }
            }

            solParcial = restoStack (solParcial);
            visitado[actual.x][actual.y] = 0;
     }
}
```

Función auxilliar para imprimir un Stack.

```
void imprimirStack (Stack* st)
{
     if (!esVacioStack (st))
     {
            Coord* act = primeroStack (st);
            imprimirStack (restoStack (st));
            printf("(%d, %d) ", act->x, act->y);
     }
}
```

El programa principal es el siguiente:

```
int main()
{
      // Creo e inicializo el mapa
      char ** mapa = crearMapa ();
      int** visitado = new int*[N];
      int i, j;
      for (i=0; i < N; i++)
      {
            visitado[i] = new int[M];
            for (j=0; j < M; j++)
                  visitado[i][j] = 0;
      }

      // movs = Stack de movimientos posibles
      // camino = Solucion parcial
      Stack* movs = crearStack (), * camino = crearStack ();
      Coord actual, c;
      c.x = 0; c.y = 1; agregarStack (movs, c);    //mov. a la derecha
      c.x = 0; c.y = -1; agregarStack (movs, c);   //mov. a la izquierda
      c.x = 1; c.y = 0; agregarStack (movs, c);    //mov. hacia abajo
      c.x = -1; c.y = 0; agregarStack (movs, c);   //mov. hacia arriba
      actual.x = 3; actual.y = 0;                  //fun. de poda,2nda rest. explicita

      solucionLaberinto (mapa, actual, visitado, camino, movs);

      // destruyo estructuras auxiliares
      destruirStack (movs);
      destruirStack (camino);
      destruirMapa (mapa);
      for (i = 0; i < N; i++)
            delete [] visitado[i];
      delete [] visitado;
      return 0;
}
```

EJERCICIO 11 (C)

Se tiene n personas a las cuales se les desea asignar n trabajos. El costo de asignar el hombre i al trabajo j es Costo[i,j]. Escribir un algoritmo basado en backtracking que minimice el costo total de las asignaciones.

Solución:

Formalización:

Forma de la solución:

- Tupla de largo fijo ***n***, de la forma <w_0, w_1, … , w_{n-1}>, donde w_i corresponde al identificador del trabajo asignado a la persona ***i***, $\forall i \;\; 0 \leq i < n$.

Restricciones explícitas:

- $w_i = j \Rightarrow 0 \leq j < n, \; \forall i \;\; 0 \leq i < n$

Restricciones implícitas:

- $w_i <> w_j \; , i \neq j, \; \forall i, 0 \leq i < n \;\; y \;\; 0 \leq j < n$

Función Objetivo:

La función objetivo es f = min (costoTupla(tupla)), donde 'costoTupla' es la suma de los costos de las asignaciones incluidas en la tupla solución t = <w_0, w_1, … , w_{n-1}> :

$$\text{costoTupla (t)} = \sum_{0}^{n-1} Costo[i, w_i]$$

Es decir, se busca $\min_{t \in T}$ (costoTupla (t)) , donde T = {t = <w_0, w_1, … , w_{n-1}> / t es solución}

```
/*
 * Algoritmo que resuelve el problema de minimizar los costos
    de la asignacion de trabajos
 *costo            -matriz de Costo[persona][trabajo]
 *solParcial       -tupla solucion hasta el momento
 *costoParcial     -costo de las asignaciones de la solucion parcial
 *mejorSol         -mejor tupla solucion conocida hasta el momento
 *mejorCosto       -costo de la mejor tupla solucion
 *pos              -posicion a asignar en la tupla, (persona)
 *trab             -trabajo a asignar
 *trab_asig        -vector que registra los trabajos ya asignados
  */
void solucionAsignar (int** costo,int * solParcial,int costoParcial, int * mejorSol,int
mejorCosto, int pos, int trab, bool* trab_asig){

    if (!trab_asig[trab]){                           //fun. de poda, restr. implicita
        solParcial[pos]= trab;
        trab_asig[trab]= true;
        costoParcial += costo[pos][trab];
        if (costoParcial < mejorCosto) {
                if ((pos==N-1)) {                    //fun. de poda, restr. explicita
                        mejorCosto=costoParcial;
                        copiar(mejorSol, solParcial);    //actualiza mejorSol
                }
                else {
                        for(sig_trab=0; i<n; i++){       //fun. de poda, restr. explicita
                                solucionAsignar  (costo,  solParcial,  costoParcial,  mejorSol,
mejorCosto, pos+1, sig_trab, trab_asig);
                        }

                }
        }
        trab_asig[trab] = false;
    }
}


// N es dato

int main()
{
        // Creacion e incializacion de Costos
        int ** costo = crearCostos();
        int* solParcial = new int[N];
        int* mejorSol = new int[N];
        bool* trab_asig = new bool[N];
        int i,t;
        for (i=0; i < n; i++){
                trab_asig[i] = false;
                solParcial[i] = -1;
                mejorSol[i] = -1;
        }
        for (t=0; t < n; t++){
                solucionAsignar   (costo,   solParcial,   0,   mejorSol,   MAX_INT,   0,   t,
trab_asig);
        }
        // destruyo estructuras auxiliares
        destruir(costo);
        delete [] solParcial;
        delete [] mejorSol;
        delete [] trab_asig;
        return 0;
}
```