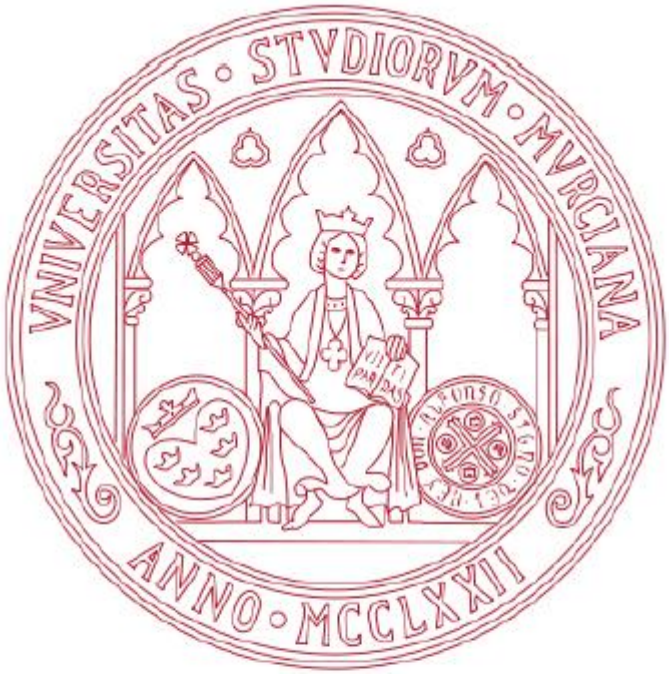


Universidad de Murcia

Facultad de Informática

PRÁCTICA DE AVANCE RÁPIDO Y BACKTRACKING

Algoritmos y Estructuras de Datos II

Alumnos:

Martín Piñas Ayala 23310083-C Grupo 2,1 martin.pinas@um.es
Mario Antelo Ribera Y1106205-B Grupo 3,1 mario.antelo@um.es

Cuenta Usada: G2_17

Profesor: Joaquín Cervera López Correo: jcervera@um.es

Índice

Lista de problemas resueltos

Los ejercicios asignados después de usar la fórmula dada:
(1106205+ 23310083 módulo 25) + 1 = 14.

14 I_AR + A_Ba

Avance Rápido

El ejercicio I de Avance Rápido corresponde con el enunciado de "Organización en Clase" cuyo objetivo es el emparejamiento de alumnos consiguiendo el máximo beneficio.

Contexto

La asignación de los puestos a los alumnos en una clase es uno de los problemas más complicados a los que se enfrentan los profesores de secundaria, pues hay que tener en cuenta las relaciones personales de los alumnos (la amistad entre ellos) pero también la compenetración en el trabajo. En este problema se trata de agrupar los alumnos de dos en dos, pues suponemos que los pupitres son de dos plazas. No tendremos en cuenta la posición de los pupitres en la clase y supondremos que lo único que se pretende es agrupar a los alumnos de dos en dos (si el número es impar uno quedará sólo en un pupitre) intentando maximizar la amistad y al mismo tiempo la compenetración en el trabajo de los alumnos que sentamos juntos.

El Problema

Escribir un programa que realice una agrupación de alumnos de dos en dos maximizando la suma de los productos del grado de amistad y la compenetración en el trabajo de alumnos que se agrupan juntos. Se dispone para ello de una matriz de *amistad* y otra de *trabajo*, donde se guardan los grados de amistad y de compenetración, que pueden no ser recíprocos, por lo que las matrices no son simétricas. La amistad y la compenetración de un alumno consigo mismo se almacenarán como cero. En cada pupitre se suman los grados de amistad de los alumnos en el pupitre, y se suman los grados de compenetración, y se multiplican los resultados de las dos sumas. El valor de beneficio es la suma de los productos obtenidos en todos los pupitres (sin incluir el posible pupitre donde sólo haya un alumno).

Suponer, por ejemplo, un caso donde el número de alumnos $N = 3$. Sean las matrices de amistad y trabajo las siguientes:

Amistad	*Trabajo*
056	053
403	302
210	150

Si numeramos los alumnos con 0, 1 y 2, la agrupación <0, 1> tendrá valor 72 (que sale de: (5+4)*(5+3)); la <0, 2> valor 32; y la <1, 2> valor 28, por lo que la mejor opción es sentar a los alumnos 0 y 1 juntos y dejar al 2 sólo.

Habrá que resolver el problema con avance rápido. La solución no tiene por qué ser la óptima, y se admitirán soluciones que no empeoren en más de un 30% de las obtenidas con el programa de los profesores (en el ejemplo la única admisible será la de la agrupación <0, 1>).

Se considerará que el número máximo de alumnos es 100.

Mooshak

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
439	2150:33:09		G2 G2 17	I_AR	C++	0 Accepted	final
438	2150:15:40		G2 G2 17	I_AR	C++	0 Wrong Answer	final
437	2150:14:46		G2 G2 17	I_AR	C++	0 Compile Time Error	final
436	2150:14:22		G2 G2 17	I_AR	C++	0 Compile Time Error	final
398	2001:04:05		G2 G2 17	I_AR	C++	0 Wrong Answer	final
397	2000:33:55		G2 G2 17	I_AR	C++	0 Wrong Answer	final
378	1180:42:45		G2 G2 17	I_AR	C++	0 Wrong Answer	final
377	1180:39:24		G2 G2 17	I_AR	C++	0 Wrong Answer	final
374	1178:25:50		G2 G2 17	I_AR	C++	0 Wrong Answer	final
356	1161:06:21		G2 G2 17	I_AR	C++	0 Wrong Answer	final
348	1159:19:32		G2 G2 17	I_AR	C++	0 Wrong Answer	final

Backtracking

En el caso del ejercicio A de Backtracking, corresponde con “la asignación de Tareas a trabajadores”, intentando maximizar el beneficio en la asignación de las tareas.

Contexto

En una empresa se dispone de varios trabajadores para realizar una serie de trabajos. Cada trabajador tiene una cierta capacidad de realizar trabajos y puede que haya trabajos que no sabe resolver. Vuestro trabajo consiste en decidir los trabajos a realizar por cada trabajador teniendo en cuenta su capacidad máxima de trabajo y los trabajos que no sabe cómo resolver. Se pretende maximizar el beneficio en la realización de los trabajos asignados, pues la habilidad en la realización de los trabajos por los trabajadores varía.

El Problema

Tenemos nt trabajos que hay que asignar a nw trabajadores. Cada trabajador i tiene una capacidad c(i), que indica el número máximo de tareas que se le pueden asignar. Si el trabajador i realiza el trabajo j, se obtiene un beneficio representado por una entrada de una tabla b(i,j). Algunos trabajadores no saben resolver algunas tareas, lo que se representa en la tabla con b(i,j)=0. Se trata de obtener de todas las posibles asignaciones de trabajos a trabajadores que cumplen las restricciones (no se asigna un trabajo a un trabajador si no lo sabe hacer, y el número de trabajos asignado a un trabajador no excede su capacidad) la que maximiza el beneficio. El beneficio de una asignación es la suma de los beneficios de realización de los nt trabajos por los trabajadores a los que se asignan. Solo son soluciones válidas aquellas en que se asignan todos los trabajos. Cuando no haya solución válida la respuesta será 0.

Mooshak

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado
564	2102:52:25		G2 G2 17	A_Ba	C++	0 Accepted
563	2102:49:16		G2 G2 17	A_Ba	C++	0 Accepted
483	1827:56:53		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
467	1766:40:54		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
466	1766:38:55		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
465	1766:35:28		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
464	1766:29:26		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
463	1766:28:42		G2 G2 17	A_Ba	C++	0 Compile Time Error
462	1766:16:08		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
452	1760:18:36		G2 G2 17	A_Ba	C++	0 Runtime Error
445	1662:46:56		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
444	1662:02:58		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
443	1600:02:24		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded
442	1599:58:07		G2 G2 17	A_Ba	C++	0 Wrong Answer
441	1599:46:53		G2 G2 17	A_Ba	C++	0 Time Limit Exceeded

El problema fundamental ha sido el tiempo en que el problema resuelve el problema.

Este problema fue resuelto añadiendo podas en la función criterio mediante cotas superiores de cada nodo.

Además se ha modificado la generación de nodos, el criterio para avanzar al siguiente nivel y las restricciones de si hay más hermanos.

Avance rápido

Pseudocódigo y explicación del algoritmo

En este apartado se pide mostrar el Pseudocódigo del algoritmo creado, pero primero queremos explicar el planteamiento que hemos realizado.

Primero escogemos un alumno, y buscamos un compañero a ese alumno entre todos los compañeros que se encuentran de pie. Para cada uno de ellos se calculará el beneficio que se obtiene al sentarlos juntos en un pupitre, y nos quedaremos con el compañero con el que mayor beneficio se ha obtenido. Se sentarán el alumno que buscaba compañero, y el compañero elegido en un pupitre, y ya no podrán ser candidatos a ser compañeros de otros alumnos. Es decir, una vez sentados no podrán ser reasignados.

Para la creación de nuestro algoritmo debemos explicar las funciones básicas que creemos pertinente:

Funciones básicas

Variables

alumnoSolo::: número de alumnos que no han sido sentados todavía.

numAlumnos::: número de alumnos.

Solución::: array donde se almacenan las parejas de alumnos sentados.

C::: array con los posibles candidatos de alumnos, se indica el estado de "cogido/ocupado".

Compañero::: el compañero de cada alumno.

Amistad::: matriz contenedora de los valores de amistad entre los alumnos.

Trabajo::: matriz contenedora de los valores de trabajo entre los alumnos.

Funciones

Voraz::: es la función principal encargada de elegir alumnos, que se encuentren de pie, y asignarle un compañero a través de la función Factible.

Factible::: el objetivo de esta función es la de emparejar un alumno con otro intentando conseguir el mayor beneficio al sentarlos juntos. Comprueba que algún alumno esté de pie y si quedan varios, irá comprobando los beneficios de cada uno y al final seleccionará al de mayor orden.

Pseudocódigo

```
Voraz () {
        Mientras ( AlumnosSolo != 0) {
                Alumno = seleccionar ();
                C:= C-{Alumno};
                AlumnosSolo:= AlumnosSolo-1;
                Factible (alumno);
        }
}

Factible (alumno) {
        Beneficio:=0;
        MayorBeneficio:=0;
        MejorCompanero = seleccionar ();

        Si (No_Alumnos_De_Pie) entonces {}
        Si no {
                Para cada alumno de pie {
                        Beneficio = CalcularBeneficio (alumno, alumnoDePie);
                        Si (Beneficio > MayorBeneficio) {
                                MejorCompanero:=alumnoDePie;
                                MayorBeneficio:=Beneficio;
                        }
                }
                C:=C-{alumnoDePie}
                Solución:= S+ {alumno, alumnoDePie}
                AlumnosSinPupitre:=AlumnosSinPupitre-1;
        }
}
```

El punto de partida será con una solución vacía, S, a la cuál iremos añadiendo parejas de alumnos que están sentándose. Tendremos N/2 beneficio, pues tenemos 2 alumnos sentados por pupitre.

El emparejamiento empieza con el primer alumno. Se busca al compañero idóneo entre los que estén de pie y haciendo que se sienten juntos. Esta operación se realizara hasta sentar a todos los alumnos, pudiendo tener compañero o estar solo, ya que puede darse el caso en que el número de alumnos sea impar y entonces habrá un alumno sin compañero, al cual sentaremos solo.

Programación del algoritmo

```
#define MAX_ALUMNOS 100

//////////////////////////////////////////////////////
//////////      VARIABLES GLOBALES      ///////////////
//////////////////////////////////////////////////////

struct pupitre{
      int alumno;
      int companero;
};

pupitre Solucion[MAX_ALUMNOS/2];

int alumnosSinPupitre;   //Entero que va a contener los alumnos que todavía no han sido sentados
int NAlumnos;          //Entero que va a contener el numero de alumnos

bool C[MAX_ALUMNOS];                //Array de booleanos que va a contener los alumnos candidatos(false si el alumno aun no ha sido sentado)
int Companero[MAX_ALUMNOS];          //Array que va a contener el compañero de cada alumno(Inicializado a -1, porque 0 es un alumno)

int Amistad[MAX_ALUMNOS][MAX_ALUMNOS]; // Matriz que contiene los valores de amistad de los alumnos
int Trabajo[MAX_ALUMNOS][MAX_ALUMNOS]; // Matriz que contiene los valores de trabajo de los alumnos

int indiceSolucion;
int beneficioTotal;
//////////////////////////////////////////////////////
//////////      FUNCIONES DEL PROGRAMA      ///////////
//////////////////////////////////////////////////////

/*Funcion que va a inicializar las variables globales*/
void resetearVariables(){
    alumnosSinPupitre = 0;
    indiceSolucion = 0;
    beneficioTotal = 0;


    for(int i = 0; i < MAX_ALUMNOS/2; i++){
          pupitre p;
          p.alumno = 0;
          p.companero = 0;
        Solucion[i] = p;
    }

    for(int i = 0; i < MAX_ALUMNOS; i++){
          C[i] = 0;
          Companero[i] = -1;    //No lo ponemos a 0 porque el 0 es un compañero
```

```
    }

    for(int i = 0; i < MAX_ALUMNOS; i++){
        for(int j = 0; j < MAX_ALUMNOS; j++){
            Amistad[i][j] = 0;
            Trabajo[i][j] = 0;
        }
    }
}

//Metodo que va a introducir los valores de las variables(Matrices, arrays,...)
void setVariables(int nAlumnos){

    alumnosSinPupitre = NAlumnos;

    //Matriz Amistad
    for(int i = 0; i < nAlumnos; i++){
        for(int j = 0; j < NAlumnos; j++){
            if(j == i){         //La diagonal de las matrices es 0
                Amistad[i][j] = 0;
            }
            else{
                int amistad = 0;
                cin >> amistad;
                Amistad[i][j] = amistad;
            }
        }
    }

    //Matriz Trabajo
    for(int i = 0; i < nAlumnos; i++){
        for(int j = 0; j < NAlumnos; j++){
            if(j == i){         //La diagonal de las matrices es 0
                Trabajo[i][j] = 0;
            }
            else{
                int trabajo = 0;
                cin >> trabajo;
                Trabajo[i][j] = trabajo;
            }
        }
    }

}


/*La función seleccionar se va a encargar de escoger, de entre los alumnos candidatos(C),
  el primer alumno que encontremos que no esté sentado*/
int seleccionar(){
    int i = 0;
    for(i = 0; i < NAlumnos-1 && C[i] == true; i++){}
```

```
    if(i == NAlumnos-1 && C[i] == true) i = -1;      //Si no quedan mas alumnos de pie, se devuelve
-1
    return i;
}

/* La funcion factible se va a encargar de, dado un alumno, buscar todos los compañeros que no
hayan sido
   ya sentados, y calcular el beneficio que obtendría con cada alumno. Vamos a ir modificando el
compañero
   que asignaremos si vamos encontrando un compañero con el que se obtenga mayor beneficio.
*/
void factible(int alumno){

    int mejorCompanero;
    int beneficio = 0;
    int mayorBeneficio = 0;

    int companero = seleccionar(); //Conseguimos el compañero por el que empezar a buscar
    mejorCompanero = companero;

    //Si no quedan mas alumnos de pie
    if(companero == -1){
            Companero[alumno] = -1;
    }
    else{

       for(; companero < NAlumnos; companero++){

            if(C[companero] == false){
                beneficio = ( Amistad[alumno][companero] + Amistad[companero][alumno] )
                            * ( Trabajo[alumno][companero] + Trabajo[companero][alumno] )  ;

                if(beneficio >= mayorBeneficio){              //Si el beneficio que obtenemos con el
compañero que estamos tratando es mayor que
                                                    //el beneficio que teniamos, actualizaremos el
compañero y el beneficio maximo
                        mejorCompanero = companero;
                        mayorBeneficio = beneficio;
                }
            }

       }

        C[mejorCompanero] = true; //Sentamos al mejor compañero que hayamos encontrado



        beneficioTotal += mayorBeneficio;

        alumnosSinPupitre--;

    }
```

```
    //saque esto del else
     pupitre p;
     p.alumno = alumno;
     p.companero = mejorCompanero;
     Solucion[indiceSolucion] = p;
}

void printSolucion(){

    //bool escogidos[NAlumnos];
    int alumno;
    int companero;
    cout << beneficioTotal << endl;
    int cant_empa = 0;

    cant_empa = NAlumnos/2;

    //cambie toda la impresion ahora todos los resultados estan en la solucion
    if(NAlumnos%2 != 0)
         cant_empa = cant_empa+1;

    //cout << NAlumnos << "-- > " << cant_empa << endl;
    for(int i = 0 ; i < cant_empa; i++){
          alumno = Solucion[i].alumno;
          cout << alumno << " ";
          if(Solucion[i].companero != -1){
                companero = Solucion[i].companero;
                cout << companero << " ";
          }
    }
    cout << endl;
}

void voraz(){
    while(alumnosSinPupitre > 0 ){

          int alumno = seleccionar();  //Se escoge, de entre C, el primer alumno que no esté sentado

          C[alumno] = true;          //El alumno se elimina del conjunto de candidatos
          alumnosSinPupitre--;

          factible(alumno);          //Se busca un companero al alumno

          indiceSolucion++;

    }
    printSolucion();

}

int main(void){
   int numeroCasos;
```

```
    cin >> numeroCasos;
    while(numeroCasos >0){

            resetearVariables();
            cin >> NAlumnos;
            setVariables(NAlumnos);
            voraz();

            numeroCasos--;

    }
    return 0;
}
```

Estudio teórico del tiempo de ejecución del algoritmo

Tendremos un bucle en el algoritmo voraz que se repetirá con N alumnos/2 veces, es decir, tantas ocasiones como parejas de alumnos se formaran.

Se seleccionará al alumno mediante el método seleccionar (), el cuál buscará el primer alumno que esté de pie en el array e indicará qué alumnos están de pies o sentados. Por tanto el método seleccionar () Є O(n). Lo siguiente, es buscar un compañero al alumno seleccionado. Usando el método factible (), se buscará todos los compañeros que estén libres, la búsqueda será en base al alumno, así el método factible () Є O(n). En definitiva, el tiempo de ejecución del algoritmo es t(n) Є O (numAlumnos/2 * (O (seleccionar ()) + O (factible ()))). Si eliminamos las constantes nos queda que t(n) Є O(n).

$$seleccionar \in O(n) \qquad factible \in O(n)$$

$$t(n) \in O\left(\frac{numAlumnos}{2} * \left(O\left(seleccionar(\ \ ) + O(factible(\ \ ))\right)\right)\right)$$

$$t(n) \in O(n)$$

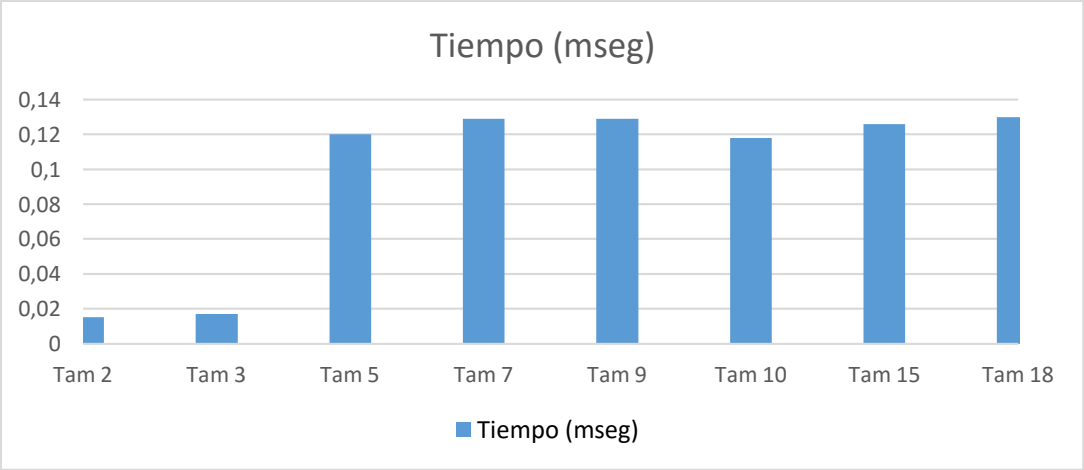
Estudio experimental del tiempo de ejecución

Hemos realizado distintas pruebas para ver el resultado de la relación de tamaño y tiempo:

Tamaño	**Tiempo** (mili segundos)
2	0,015
3	0,017
5	0,12
7	0,129
9	0,129
10	0,118
15	0,126
18	0,13

Se puede observar que el tiempo empleado por el algoritmo de avance rápido es bastante bueno, sobretodo, cuando el tamaño de los problemas es grande emplea muy poco tiempo en hallar una solución. En ocasiones el resultado no es el más óptimo pero teniendo en cuenta el tiempo empleado, se consigue que sea un buen algoritmo para hallar una solución.

Al realizar las pruebas, nos damos cuenta que este tipo de algoritmos será conveniente cuando el tamaño del problema sea muy grande.



Comparación del estudio teórico y el experimental

Una vez realizado el estudio experimental, hemos comprobado que nos aproximamos bastante en el estudio teórico del tiempo de ejecución, ya que obteníamos un tiempo lineal, y como reflejan los resultados conseguidos experimentalmente, se puede constatar que el tiempo aumenta de manera más o menos lineal.

No se observa ningún gran salto en el tiempo respecto a los aumentos del tamaño del problema.

Backtracking

Pseudocódigo y explicación del algoritmo

Antes de comenzar con la explicación de los cambios que hemos introducido en el algoritmo base, vamos a recordar las ideas fundamentales de Backtracking.

Backtracking realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Prueba todas las combinaciones. Por ello, suele resultar muy ineficiente.

En nuestro problema el árbol implícito será un árbol permutacional en el que cada nivel se decide a que trabajador se asigna la tarea actual, es decir, cada nivel del árbol representara a una tarea.

La condición de fin será que volvamos al nodo raíz o que en la primera rama encontremos una solución que sea el máximo beneficio posible.

Se recorrerá todo el árbol comprobando en cada nodo si hemos llegado a un nodo hoja, es decir, si en la solución se han evaluado todas las tareas.

Para mantener la solución más óptima mientras se recorre todo el árbol hemos utilizado las siguientes variables:

- ***Voa***: es el beneficio más óptimo encontrado
- ***S***: contiene los trabajadores seleccionados para todas las tareas

Además hemos utilizado algunas variables extras para mantener la información en cada nodo:

- ***Trab_asig***: es un vector de tamaño n (trabajadores) que permite mantener la información sobre los la cantidad de tareas asignadas a un trabajador.
- ***Cotas_Superior***: vector que mantiene el beneficio máximo que podemos obtener desde el nodo actual.

El algoritmo es similar a un esquema de maximización en la asignación de tareas.

Representamos un ejemplo de los valores que se guardan en el vector de cotas superiores:

5	1	2	1
2	3	3	5
2	5	3	2
2	3	5	3

Beneficio de los trabajadores

20	15	10	5

Cota Superior

Pseudocódigo

```
backtracking(……..){
        while (tarea >= tarea_Raiz)
                si generamos_nodo
                        si es_solucion Y beneficio_actual es mayor a beneficio_Optimo
                            beneficio_Optimo = beneficio_actual;
                            si beneficio_Optimo es igual al beneficio_mayor
                                regresar beneficio_Optimo;
                        finsi
                        si podemos_continuar
                                nivel++;
                        sino
                                si no mas_hermanos
                                        retrocedemos_tarea
                        finsi
                sino
                        retrocedemos_tarea
return beneficio_Optimo;
```

Procedemos a explicar el algoritmo descrito anteriormente:

- Generar_nodo: esta es una función que lo que realiza es la asignación de un trabajador a la tarea actual.
  Cada asignación cumple una serie de restricciones como que el beneficio del trabajador sea mayor o igual al del trabajador anterior (con esto construimos un árbol permutacional).
  Que el beneficio de cada trabajador sea distinto de 0.
  Que el trabajador que se asigna no haya cumplido su cupo de tareas.
- Es_solucion: el algoritmo llegara a una solución cuando se haya asignado todas las tareas.
- Podemos_continuar: aquí se evalúa si podemos llegar a una solución óptima desde el nodo actual.
- Mas_hermanos: comprueba si por cada tarea hemos evaluados todos los trabajadores.

Programación del algoritmo

```
/*
* Función no Recursiva del algoritmo backtracking
* encontrara una solucion y la imprimira por pantalla
* recibira 4 parametros
*    - Matriz con la capacidades de cada trabajador
*    - Número de trabajadores
*    - Número de tareas a asignar
*    - vector con las cotas de cada nodo.
* no devuelve nada
*/
int backtracking(vector<Trabajador> bw, int nw, int nt, vector<int> cotaSuperior){
        //vector solucion y vector de asignaciones a tarea.
        vector<int> S, trab_asig;

        S.assign(nt, -1);
        trab_asig.assign(nw, 0);

        //nivel por el que empieza el algoritmo
        int nivel=0;

        int voa = 0; //beneficio optimo
        int bact =0; //beneficio actual

        while (nivel >= 0) {
                //intentamos generar un nodo
                if(generar(nivel, S, bw, trab_asig, bact)  ){
                                //comprobamos si el nodo actual es una solucion
                                if (solucion(nivel, nt) && (bact > voa) ){
                                        voa = bact;
                                        if (cotaSuperior.at(0) == bact)
                                                return bact;
                                }
                                //evaluamos la siguiente rama para comprobar si podemos continuar
por ahi
                                if (criterio(nivel, nt,voa, bact, cotaSuperior))
                                        nivel++;
                                else {
                                        //si no podemos continuar por la misma
                                        // se comprueba si se puede generar un nodo hermano
                                        while(nivel >= 0 && !masHermanos(S, nivel,nw) )
                                                retroceder(nivel, S, trab_asig,bw,bact);

                                }
                }else {
                                //si no asignamos a ningun trabajador se vuelve a la tarea anterior
                                S[nivel] = -1;
        nivel--;
                }
        }
        return voa;
}
```

```
/*
* Funcion Generar
* funcion que genera un nodo con una posible solucion.
* la posible solucion sera la asignacion de un trabajador para esa tarea
* retorna true o false, dependiendo de si se genera el nodo
*
*/
bool generar(int nivel, vector<int> &s, vector<Trabajador> &bw, vector<int> &asigna, int
&bact ){
                int valor ;
                int nw = bw.size();
                valor = s[nivel];
        s[nivel]++;
        int capAnt =0;
        //si se ha asignado un trabajador lo desasignamos
        if(valor != -1){
                capAnt = bw.at(valor).getCapacidad(nivel);
           bact -= capAnt;
           asigna.at(valor)--;
        }
        //buscamos un trabajador para la tarea actual(nivel)
        while((s[nivel] < nw)){
                int cap = bw.at(s[nivel]).getCapacidad(nivel);
           if( cap > 0  && cap >= capAnt && asigna.at(s[nivel]) <
bw.at(s[nivel]).get_maxTrab()){
                asigna.at(s[nivel])++;
                bact += cap;
                return true;
           }else{
                s[nivel]++;
           }
        }
                return false;
}

/*
* Funcion retroceder
* se retrocede a la tarea anterior
*/
void retroceder(int &nivel, vector<int> &s, vector<int> &asig, vector<Trabajador> &bw,int
&bact){
        int valor = s[nivel];
        s[nivel]=-1;

        nivel--;
        if(valor >= 0){
                asig.at(valor)--;
                bact -= bw.at(valor).getCapacidad(nivel+1);
        }
}
```

```
/*
* Funcion Solucion
* llegaremos a una solucion cuando lleguemos a la ultima tarea
*/
bool solucion(int nivel, int nivelMax){
	if (nivel == nivelMax-1)
			return true;
	return false;
}

/*
* Funcion Criterio
* esta funcion es para comprobar si podemos continuar por el siguiente nodo
* se ha agregado podas de un nodo futuro con las cotas superiores
*/
bool criterio(int nt, int ntmax, int voa, int bact, vector<int> cotaSuperior){
	if(nt < ntmax-1){
		if (voa < (bact + cotaSuperior[nt+1]))
			return true;

	}
	return false;
}
/*
* Funcion masHermanos
* va a comprobar si para una misma tarea
* se puede desplegar un nodo derecho
* devuelve true si podemos desplegar un nodo derecho
*/
bool masHermanos(vector<int> s, int nivel, int nw){
   return (s[nivel] < nw-1);

}
```

Estudio teórico del tiempo de ejecución del algoritmo

Para realizar el estudio teórico del tiempo de ejecución de nuestro algoritmo, hemos calculado los tiempos para el mejor y peor caso

El mejor caso sería que se encontrara la solución óptima en la primera rama del árbol de búsqueda, o lo que es lo mismo, en las primeras nt iteraciones. Las funciones (solución, generar…) no afectan al tiempo ya que se son de orden constante para este caso.

El orden de ejecución teórico de nuestro algoritmo sería:

$$t(n) \in O(nt)$$

En cuanto al peor caso, sería que se tuviera que recorrer todo el árbol de búsqueda para encontrar la solución óptima. Por lo tanto el orden del algoritmo sería de:

$$t(n) \in O\left(nt^{nw}\right)$$

Este orden sale porque tenemos que recorrer todas las tareas y para cada una de ellas, todos los trabajadores para ver la asignación.

En este caso, la función generar no es de orden constante, sino de

$$generar \in O(n)$$

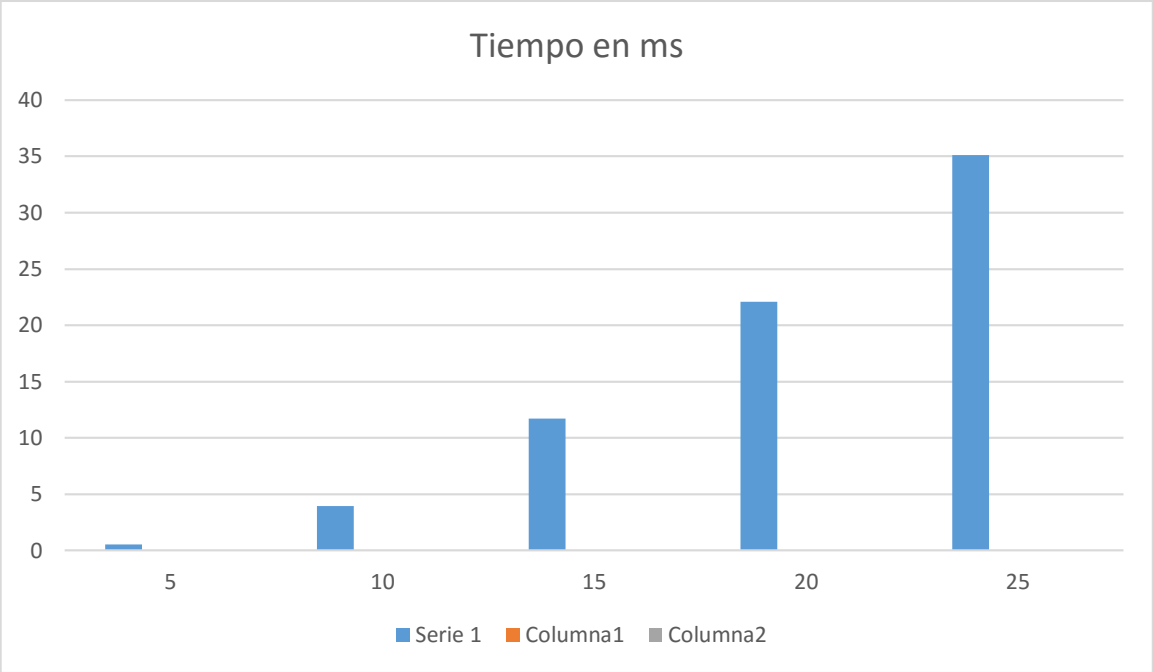
Ya que se recorre todos los nodos hermanos para generar uno válido. Si lo hiciéramos de orden constante esta función, es decir, generando el nodo al que le toque en ese momento directamente, generaríamos nodos de más que más adelante se descartarían en la función criterio.

Estudio experimental del tiempo de ejecución

Hemos realizado distinta pruebas donde para un mismo tamaño de tareas variamos los trabajadores que pueden ser asignados

Trabajadores	**Tiempo** **(mili segundos)**
5	0.55
10	3.94
15	11.69
20	22.08
25	35.11

Se puede observar que el tiempo empleado por el algoritmo de backtraking crece exponencialmente al tener más trabajadores para asignar para una misma tarea.



Comparación del estudio teórico y el experimental

Una vez realizado el estudio experimental, hemos comprobado que nos aproximamos bastante en el estudio teórico del tiempo de ejecución, ya que obteníamos un tiempo exponencial con respecto al número de trabajadores.