ALGORITMOS Y ESTRUCTURAS DE DATOS II

PRACTICA 2: AVANCE RÁPIDO Y BACKTRACKING

Subgrupo 3.3 – Facultad de Ingeniería Informática – Universidad de Murcia

Autores:

José Ángel García Marín – 49478872F

Juan Rondán Ramos – 48854653P

Profesor:

Francisco José Montoya Dato

Cuenta Mooshak: G3-51

Contenido

Problemas a realizar

(49478872 + 48854653) % 25 +1 = 1

Por tanto, la combinación de ejercicios a realizar es la 1:

- Backtracking: A
- Avance rápido: C

Avance Rápido

Envío Mooshak

El envío final del problema de avance rápido es el 553.

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
553	1058:46:51		G3 GARCIA MARIN-RONDAN RAMOS	C AR	C++	2 Accepted	final

La realización de este problema no causó demasiadas complicaciones, pues siguiendo el algoritmo visto en clase y haciendo algunas adaptaciones conseguimos obtener una buena solución al problema.

Pseudocódigo del Algoritmo

Función seleccionarElementoInicial

Esta función se usa para elegir el primer elemento que se selecciona cuando se empieza con el conjunto vacío. Su funcionamiento consiste en elegir aquel elemento con más diversidad total, para intentar escoger uno de los elementos con mayor potencial.

```
seleccionarElementoInicial(distancias: matriz de enteros)

    int mejorIndice = -1;
    int mejorSuma = -1;

    PARA (size_t i = 0) HASTA (i < distancias.size()) HACER:
        int suma = 0;

        PARA (size_t j = 0) HASTA (j < distancias.size()) HACER:
            suma += distancias[i][j];
            j++;
        FIN PARA

        SI (suma > mejorSuma) ENTONCES
            mejorSuma = suma;
            mejorIndice = i;
        FIN SI

        i++;
    FIN PARA

    DEVOLVER mejorIndice;
```

Función seleccionarMejorCandidato

Esta función corresponde a la función "seleccionar" de la plantilla para algoritmos de avance rápido vista en clase, que se encarga de seleccionar el mejor candidato actual que se puede añadir. Para ello, calcula el beneficio que tendría en la diversidad total cada uno de los candidatos y escoge aquel cuyo beneficio por seleccionarlo es mayor.

```
seleccionarMejorCandidato(distancias: matriz de enteros, candidatos, seleccionados: lista de enteros)

    int mejorElemento = -1;
    int mejorContribucion = -1;

    PARA CADA candidato EN candidatos HACER:
        int suma = 0;

        PARA CADA seleccionado EN seleccionados HACER:
            suma += distancias[candidato][seleccionado] + distancias[seleccionado][candidato];
        FIN PARA

        SI (suma > mejorContribucion) ENTONCES
            mejorContribucion = suma;
            mejorElemento = candidato;
        FIN SI
    FIN PARA

    DEVOLVER mejorElemento;
```

Función seleccionarPeor

De manera similar a la función seleccionarMejorCandidato, esta función corresponde a la función "seleccionar". Sin embargo, esta función se usa para encontrar el elemento que menos beneficio produce para poder sacarlo del conjunto de seleccionados. Para ello se calcula el beneficio en la diversidad total de cada uno de los elementos y se devuelve aquel que menos beneficio genera (y que, por tanto, sería menos perjudicial eliminar del conjunto de seleccionados).

```
seleccionarPeor(distancias: matriz de enteros, seleccionados: lista de enteros)

    int peorElemento = -1;
    int peorContribucion = infinito;

    PARA CADA candidato EN seleccionados HACER:
        int suma = 0;

        PARA CADA seleccionado EN seleccionados HACER:
            suma += distancias[candidato][seleccionado] + distancias[seleccionado][candidato];
        FIN PARA

        SI (suma < peorContribucion) ENTONCES
            peorContribucion = suma;
            peorElemento = candidato;
        FIN SI
    FIN PARA

    DEVOLVER peorElemento;
```

Función algoritmoVoraz

Esta es la función principal que, dado una matriz de distancias (o diversidad como dice el enunciado) y un entero m (número de elementos seleccionados) devuelve el conjunto de elementos seleccionados.

Para ello, actúa de diferentes maneras en funcion del tamaño de m:

-Si m es mayor que la raiz cuadrada de n*(n+1)/2 entonces selecciona todos los elementos para, a continuación, escoger los n-m elementos que menos diversidad aportan. Cuando se hace de esta manera en lugar de empezar con el conjunto vacio y añadir m elementos es porque es más rentable en tiempo de ejecución.

-Si m=0 no tiene que hacer nada puesto que la respuesta siempre será el conjunto vacío.

-En el resto de casos se comienza con el conjunto vacío, se selecciona primero el elemento con mejor expectativas y posteriormente se seleccionan los m-1 elementos que más beneficio producen en la diversidad total sucesivamente.

```
algoritmoVoraz(distancias: matriz de enteros, m: entero, seleccionados: lista de enteros)

    int n = distancias.size();

    SI (m > redondear(raíz((n * (n + 1)) / 2))) ENTONCES

        seleccionados.resize(n);

        PARA (i = 0) HASTA (i < n) HACER:
            seleccionados[i] = i;
            i++;
        FIN PARA

        MIENTRAS (seleccionados.size() > m) HACER:
            int siguiente = seleccionarPeor(distancias, seleccionados);
                        seleccionados.erase(remove(seleccionados.begin(), seleccionados.end(), siguiente), seleccionados.end());
        FIN MIENTRAS

    SI_NO SI (m == 0) ENTONCES
        // No hacer nada

    SI_NO

        vector<int> candidatos = lista de tamaño n;

        PARA (i = 0) HASTA (i < n) HACER:
            candidatos[i] = i;
            i++;
        FIN PARA
```

```
        int inicial = seleccionarElementoInicial(distancias);
        seleccionados.push_back(inicial);
        candidatos.erase(remove(candidatos.begin(), candidatos.end(), inicial), candidatos.end());


        MIENTRAS (seleccionados.size() < m Y !candidatos.empty()) HACER:
            int siguiente = seleccionarMejorCandidato(distancias, candidatos, seleccionados);

            SI (siguiente == -1) ENTONCES
                DEVOLVER falso;
            FIN SI

            seleccionados.push_back(siguiente);
            candidatos.erase(remove(candidatos.begin(), candidatos.end(), siguiente), candidatos.end());

        FIN MIENTRAS

    FIN SI

    DEVOLVER (seleccionados.size() == m);
```

Estudio Teórico

Para analizar el tiempo de ejecución del algoritmo algoritmoVoraz en función del tamaño n de la matriz de distancias y del parámetro m (número de elementos a seleccionar), desglosamos los componentes clave y analizamos tres casos: mejor caso, caso promedio y peor caso.

Recordamos que n es el número total de elementos y m el número de elementos a seleccionar ($m \leq n$).

Resumen de funciones auxiliares (costos relevantes):

seleccionarElementoInicial → $O(n^2)$

seleccionarMejorCandidato y seleccionarPeor → $O(n^2)$ (funciones llamadas hasta m veces)

erase(remove(...)) (de la clase vector) → O(n) en el peor caso por operación

NOTA: Téngase en cuenta que probablemente usar arrays simples en lugar vector hubiese sido más eficiente, no obstante, utilizamos la clase vector puesto que permite ver con más claridad el uso del algoritmo visto en clase.

En el **mejor caso**, que se daría cuando m=0 tenemos un tiempo de orden constante O(1). Esto se debe a que la solución siempre va a ser el conjunto vacío puesto que no habrá que añadir ningún elemento al conjunto de seleccionados.

En el **peor caso**, cuando m= n/1,41+ 1, es cuando el coste es mayor. En un principio, intuíamos que el peor caso estaría m=n/2 (o n/2+1) ya que es cuando más elementos hay que añadir (o eliminar) y más llamadas había que hacer a las funciones seleccionarMejorElemento y seleccionarPeor. No obstante, al realizar el estudio experimental, nos dimos cuenta de que no era así y tuvimos que reformular la hipótesis. Se debe a que eliminar cierto número de elementos menor a n/2 es más costoso (siempre hablando en tiempo de ejecución) que añadirlos debido a que, aunque se llama a las funciones seleccionarMejorElemento y seleccionarPeor es número de veces, cada llamada individual a seleccionarPeor es más costosa puesto que se llama con parámetros más grandes. Dado que el coste de estas funciones depende del número de candidatos y del número de seleccionados, observamos que el número de candidatos es similar en ambos casos. Mientras tanto, el número de elementos seleccionados con el que se llama a seleccionarMejorElemento se incrementa desde 1 a m mientras que, el número de elementos seleccionados con el que se llama a seleccionarPeor de decrementa desde n a m, siendo el número de elementos claramente mayor en el segundo caso. Por tanto, será mejor empezar desde 1 hasta m incrementando en lugar de con n hasta m decrementando cuando el sumatorio de 1 hasta m no sea mayor que el sumatorio de m a n.

Esto sucede cuando $2m^2 > n(n+1)$) o, lo que es lo mismo, cuando: $m = \sqrt{\frac{n(n+1)}{2}}$.

Como se hacen m o n-m llamadas de coste $2m^2$ o n(n+1)), y en realidad el orden es de m es O(n) podemos asumir que el coste tiene orden de $O(n^3)$.

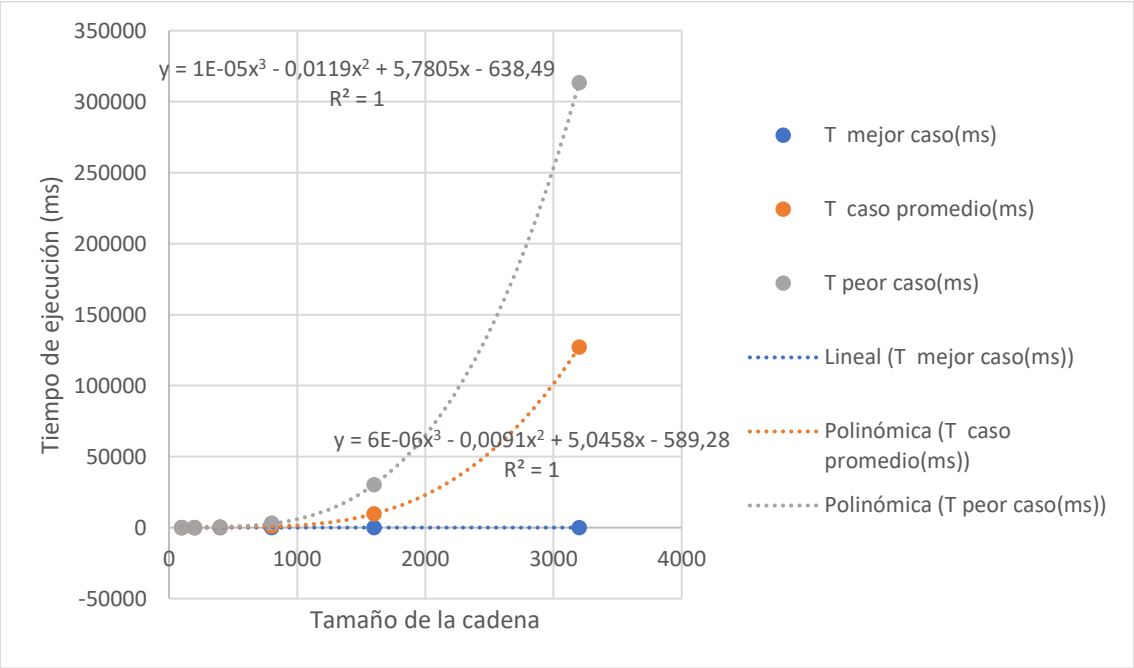
En el **caso promedio**, en coste es igualmente de $O(n^3)$, pero con una constante algo menor que en el peor caso.

NOTA: En el envío de mooshak aparece cuando m = n/1,41 puesto que consideramos que para n grandes es aproxima mucho a la raíz de ((n * (n + 1)) / 2), no obstante en los códigos que se entregan se muestra el código y las pruebas con m igual a la raíz de ((n * (n + 1)) / 2).

Estudio Experimental

Para el estudio experimental hemos generado pruebas automáticamente añadiendo funciones para la generación de matrices de distancias y hemos ido duplicando n hasta que el tiempo de ejecución ha sido demasiado alto (en este caso a partir de 3200 tardaba demasiado). Para el tiempo promedio se ha realizado una simulación puesto que hacer un estudio con un tamaño suficientemente grande hubiese sido excesivamente costoso en tiempo.

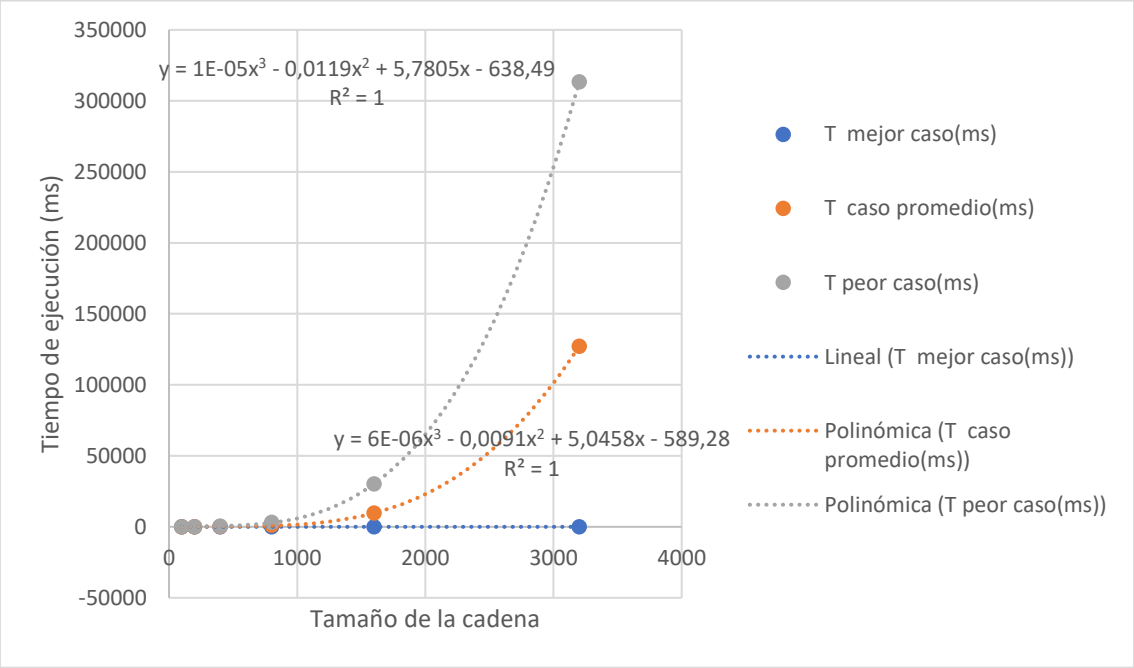
n	T mejor caso(ms)	T caso promedio(ms)	T peor caso(ms)
100	0,001	2,0706	5,321
200	0,001	15,0187	45,225
400	0,001	118,027	349,633
800	0,001	1017,27	3089,92
1600	0,001	9678,64	30240,3
3200	0,001	127152	313276



Contraste del estudio Teórico y Experimental

Para contrastar los resultados del estudio experimental de los casos peor y promedio hemos realizado una regresión polinómica de grado 3 en Excel, obteniendo para ambos $R^2=1$, lo que evidencia que los resultados se corresponden con el análisis teórico.

Para el mejor caso no es necesario realizar ninguna regresión puesto que es evidente que el orden del tiempo en el mejor caso es constante (y por tanto se corresponde al estudio teórico) puesto que hemos obtenido el mismo coste en tiempo en todas las pruebas.



Backtracking

Explicación del Algoritmo

El problema consiste en asignar un conjunto de nt tareas a nw trabajadores, de forma que se maximice el beneficio total obtenido, bajo la restricción de que cada trabajador puede realizar solo un número limitado de tareas, según su capacidad individual.

Para resolverlo, se implementa un algoritmo de backtracking con poda, que explora las posibles asignaciones de tareas a trabajadores de manera sistemática. La exploración sigue la estructura de un árbol n-ario de asignaciones, donde:

- Cada nivel del árbol representa una tarea que debe asignarse.
- Cada rama desde un nodo representa una posible asignación de esa tarea a un trabajador específico, siempre que tenga capacidad restante.
- La profundidad del árbol es igual al número de total de tareas (nt), y el número de ramas posible en cada nodo depende del número de trabajadores (nw) y de su disponibilidad en ese momento.

Esta estructura permite construir las asignaciones de forma incremental, evaluando combinaciones viables a medida que se avanza en profundidad por el árbol. En cada paso se lleva un seguimiento del beneficio acumulado y de las capacidades restantes de los trabajadores, lo que permite garantizar que las restricciones se respeten en todo momento.

Adicionalmente, se incorpora una técnica de poda por cota superior. En cada nodo, se estima el beneficio máximo que podría alcanzarse desde el estado actual hasta el final, asumiendo que las tareas restantes se asignan de forma óptima. Si esta estimación no supera el mejor resultado encontrado hasta el momento, se descarta la exploración de dicha rama, lo que reduce significativamente el número de combinaciones exploradas.

Gracias a este enfoque, el algoritmo logra reducir significativamente el número de combinaciones exploradas en muchos casos prácticos, manteniendo una estructura clara y adecuada a la naturaleza combinatoria del problema.

Envío Mooshak

El envío final del problema de backtracking es el 394.

Para este problema en concreto, no hemos tenido grandes complicaciones para su realización. Al haber trabajado en clases de teoría el funcionamiento del backtracking paso a paso de forma manual e incluso problemas muy similares, teníamos las ideas muy claras de lo que necesitábamos.

Pseudocódigo Algoritmo

Función calcularBeneficioMaxPorTrabajo

Esta función analiza la tabla de beneficios y calcula, para cada tarea, el beneficio máximo que se podría obtener si se asigna al mejor trabajador posible. El resultado es un vector de tamaño igual al número de tareas, donde cada posición contiene el valor máximo alcanzable para esa tarea. Esta información es crucial para aplicar poda por cota superior durante el backtracking, permitiendo estimar si aún es posible superar el mejor beneficio encontrado hasta el momento.

```
calcularBeneficioMaxPorTrabajo(nw, nt: entero; tabla: matriz de enteros) -> lista de enteros
    max_por_trabajo = lista de tamaño nt, inicializada a 0

    PARA (j = 0) HASTA (j < nt)  HACER:
        PARA (i = 0) HASTA (i < nw) HACER:
        max_por_trabajo[j] = max(max_por_trabajo[j], tabla_beneficios[i][j]);
            i++;
    FIN PARA
    j++;
    FIN PARA

    DEVOLVER max_por_trabajo
```

Función backtracking

Esta función explora todas las formas posibles de asignar las tareas a los trabajadores respetando sus capacidades y maximizando el beneficio total. A cada llamada se le pasa la tarea que debe asignarse en ese momento y el beneficio acumulado hasta ese momento. Para cada trabajador que aún tenga capacidad y aporte beneficio positivo a la tarea actual, se realiza una asignación temporal y se avanza a la siguiente tarea. Antes de continuar, se calcula una estimación del beneficio máximo alcanzable sumando el beneficio actual y el mayor beneficio posible por cada tarea restante. Si esta estimación no supera el mejor beneficio registrado (maxBeneficioGlobal), se descarta esa ruta y no se sigue explorando. En caso de completar todas las tareas, se actualiza el beneficio global si el actual es mejor. Esta función permite resolver el problema de forma óptima sin necesidad de explorar combinaciones que no pueden mejorar el resultado ya conocido.

```
backtracking(tareaActual, beneficioActual, nw, nt: entero;
             tabla_beneficios: matriz de enteros; capacidades_restantes, max_por_trabajo: lista de enteros)

    bool continuar = true;

    SI (tareaActual == nt) ENTONCES
        SI (beneficioActual > maxBeneficioGlobal) ENTONCES
            maxBeneficioGlobal = beneficioActual;
        FIN SI
        continuar = FALSO
    FIN SI


    SI (continuar) ENTONCES
        int beneficioPosible = beneficioActual

        PARA (i = tareaActual HASTA i < nt) HACER:
            beneficioPosible += max_por_trabajo[i];
        i++;
        FIN PARA

        SI (beneficioPosible <= maxBeneficioGlobal) ENTONCES
            continuar = FALSO;
        FIN SI
    FIN SI
```

```
    SI (continuar) ENTONCES
        PARA (trabajador = 0 HASTA trabajador < nw) HACER:
            int b = tabla_beneficios[trabajador][tareaActual];

            SI (b > 0 Y capacidades_restantes[trabajador] > 0) ENTONCES
                capacidades_restantes[trabajador]--;

                backtracking(tareaActual + 1, beneficioActual + b, nw, nt, tabla_beneficios, capacidades_restantes, max_por_trabajo);

                capacidades_restantes[trabajador]++;
            FIN SI
        trabajador+;
        FIN PARA
    FIN SI
```

Función resolverCaso

Esta función prepara y lanza el algoritmo de backtracking para un único caso de prueba. Primero, reinicia el beneficio global (maxBeneficioGlobal) para que no arrastre valores de otros casos. Luego inicializa la estructura de capacidades restantes y calcula el vector de beneficios máximos por tarea, para preparar las podas. Finalmente llama a backtraking(…) desde la tarea 0 con beneficio acumulado 0, y devuelve el mejor resultado encontrado tras explorar todas las combinaciones válidas

```
resolverCaso(nw, nt: entero; tabla_beneficios: matriz de enteros; capacidades: lista de enteros) -> entero
    maxBeneficioGlobal = 0;

    vector<int> capacidades_restantes = copia de capacidades;
    vector<int> max_por_trabajo = calcularBeneficioMaxPorTrabajo(nw, nt, tabla_beneficios);

    backtracking(0, 0, nw, nt, tabla_beneficios, capacidades_restantes, max_por_trabajo);

    DEVOLVER maxBeneficioGlobal;
```

Estudio Teórico

Orden de complejidad (peor caso)

En el peor de los casos, el algoritmo explora todas las posibles asignaciones de tareas a trabajadores. Para cada una de las nt tareas, se evalúa la posibilidad de asignarla a cualquiera de los nw trabajadores, siempre que su capacidad restante lo permita.
Si no se aplicara ninguna poda y todos los trabajadores tuvieran capacidad suficiente, el número total de asignaciones sería:

$$nw^{nt}$$

Esto se debe a que, en cada nodo de los nt niveles del árbol de decisiones pueden existir hasta nw ramas, una por cada trabajador. Por tanto, el orden en el peor caso es:

$$O(nw^{nt})$$

Esto es una complejidad exponencial respecto al número de tareas.

Efecto de las podas

Aunque el algoritmo implementa una estrategia de poda por cota superior, esta no modifica el orden de complejidad en el peor caso, que sigue siendo exponencial.

La poda consiste en calcular una estimación optimista del beneficio máximo que se podría obtener desde el estado actual hasta el final de la asignación. Para ello, se suman los beneficios máximos posibles para cada una de las tareas aún no asignadas, asumiendo que podrán asignarse a los trabajadores que aporten el mayor beneficio. Si esta estimación no supera el beneficio global máximo encontrado hasta ese momento, se descarta esa rama del árbol, ya que no podrá mejorar la solución actual.

Este mecanismo permite reducir significativamente el número de combinaciones que se exploran en la práctica, sobre todo cuando algunas tareas dan beneficios claramente mayores que otras o cuando los trabajadores tienen pocas tareas que pueden hacer. Sin embargo, no siempre funciona igual de bien: si los beneficios son muy parecidos entre sí la poda podría activarse pocas veces o incluso ninguna, por lo que el algoritmo puede acabar recorriendo todo el árbol de posibilidades igualmente.

Estudio Experimental

Con el objetivo de analizar empíricamente el comportamiento del algoritmo, se ha llevado a cabo un estudio experimental centrado en dos aspectos fundamentales: la eficiencia de la poda y la confirmación del crecimiento exponencial de ejecución respecto al tamaño de la entrada.

Para ello, se ha partido del código original aceptado en Mooshak, al cual se le ha añadido la librería *sys/time.h* para poder medir y mostrar en pantalla el tiempo de ejecución de cada caso de prueba.

Se han implementado dos versiones del algoritmo de backtracking:

- Una versión con poda (la enviada a Mooshak), que aplica una estimación por cota superior para evitar combinaciones que no pueden superar la mejor solución conocida.
- Una versión sin poda, que explora todas las asignaciones posibles respetando las restricciones de capacidad, sin ningún tipo de optimización adicional.

Además, se ha desarrollado un generador automático de casos de prueba, configurado para crear instancias en las que el número de trabajadores y tareas es el mismo (nw = nt), con el fin de simplificar la configuración de entrada y controlar mejor el crecimiento del problema. Los tamaños de los casos se especifican mediante un vector en el main, y que por ende, su longitud son los casos de prueba totales.

La comparación de tiempos entre ambas versiones del algoritmo se ha realizado utilizando casos de entrada pequeños, ya que la versión sin poda se vuelve rápidamente ineficiente. No obstante, estos resultados son suficientes para mostrar una diferencia clara entre ambos enfoques.

Finalmente se han ejecutado instancias de mayor tamaño únicamente con el algoritmo que sí incluye poda, con el fin de observar de forma más evidente el comportamiento exponencial del algoritmo a medida que aumentan los trabajadores y las tareas. Los resultados obtenidos se han representado gráficamente para visualizar mejor la evolución del tiempo de ejecución en función del tamaño del problema.

Para el estudio experimental, se ha añadido al código original aceptado por Mooshak la librería sys/time.h, utilizado para medir y mostrar en pantalla el tiempo que tarda cada caso. Posteriormente, se han realizado algunas mediciones para algunos casos de entrada, suponiendo que el número de trabajadores y tareas es el mismo, por simplicidad.

Para estos casos de entrada, se ha creado un generador de casos de prueba, en cuyo main hay un vector donde se indican el tamaño de los casos de entrada y que por ende, su longitud son los casos de prueba totales.

Y, adicionalmente, se ha implementado un algoritmo de backtracking sin poda, para hacer la comparativa de tiempos con el que si incluye poda. Se han tenido que usar casos de prueba muy pequeño debido a la tardanza del algoritmo, pero suficientes para mostrar como uno es muchísimo mejor que otro.

Comparativa: algoritmo con poda vs sin poda

En cuanto a la comparativa entre el tiempo de ejecución del algoritmo con poda y el algoritmo sin poda, se han tomado 5 medidas para diferentes casos de entrada del mismo tamaño, generados con nuestro generador de entradas, anotando el tiempo de ejecución en milisegundos (ms).

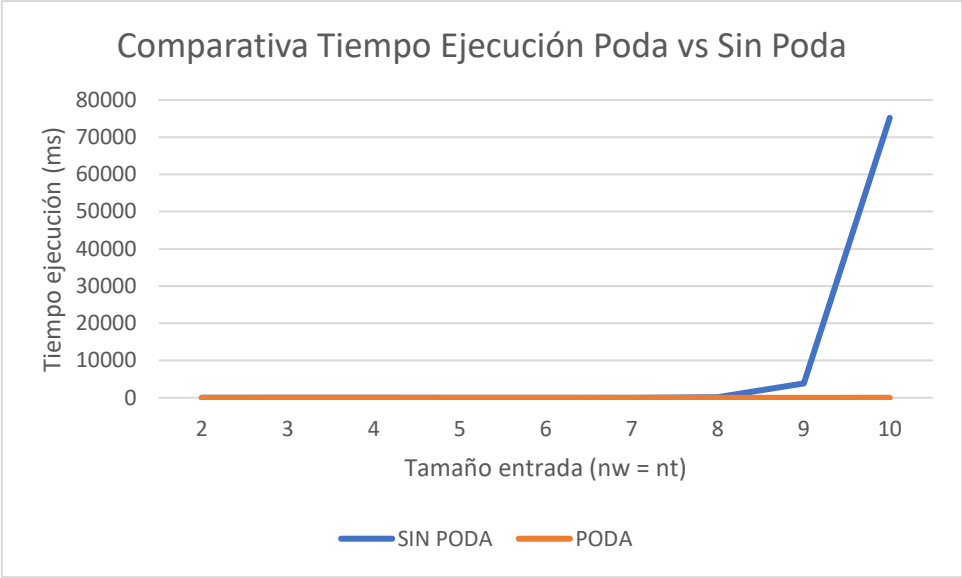
Algoritmo sin poda:

nt / nw	1º tiempo	2º tiempo	3º tiempo	4º tiempo	5º tiempo	media
2	0,187	0,179	0,048	0,045	0,082	0,1082
3	0,145	0,183	0,059	0,039	0,12	0,1092
4	0,146	0,144	0,121	0,089	0,19	0,138
5	0,185	0,177	0,181	0,173	0,181	0,1794
6	0,412	1	0,0774	0,964	1,076	0,70988
7	12,608	18,841	9,081	19,438	19,273	15,8482
8	40,457	143,66	160,95	246,63	132,9	144,9194
9	3709,45	4611,38	5235,67	3100,89	2728,31	3877,14
10	110457	91141,6	106140	33022,4	35237	75199,6

Algoritmo con poda:

nt / nw	1º tiempo	2º tiempo	3º tiempo	4º tiempo	5º tiempo	media
2	0,045	0,045	0,045	0,043	0,046	0,0448
3	0,018	0,011	0,04	0,011	0,034	0,0228
4	0,017	0,011	0,033	0,011	0,01	0,0164
5	0,017	0,013	0,039	0,013	0,011	0,0186
6	0,017	0,015	0,44	0,018	0,047	0,1074
7	0,026	0,071	0,051	0,027	0,026	0,0402
8	0,029	0,026	0,037	0,027	0,023	0,0284
9	0,028	0,102	0,049	0,07	0,042	0,0582
10	0,039	0,052	0,184	0,177	0,144	0,1192

Tan solo con echar un vistazo a las tablas, se puede observar que es totalmente inviable utilizar el algoritmo sin poda, ya que para tamaños relativamente pequeños (10), obtenemos tiempos cercanos y superiores al minuto (75s de media). Por otro lado, el algoritmo con poda, parece funcionar relativamente bien, para estos tamaños de entrada obtenemos tiempos por debajo del milisegundo. En la siguiente gráfica se muestra la comparativa:



Se puede observar cómo, a partir de cierto umbral, el algoritmo sin poda pasa a ser hasta casi 80 segundos más lento, en promedio, respecto al algoritmo con poda, dejando en evidencia su inviabilidad práctica sin mecanismos de optimización.

Al seguir obteniendo tiempos tan bajos, se ha seguido aumentado el tamaño de entrada para obtener valores más significativos y verificar el orden exponencial del que se habló en el análisis teórico.

A continuación, se muestran los resultados del tiempo de ejecución para unos tamaños del problema más grandes:

nt / nw	1º tiempo	2º tiempo	3º tiempo	4º tiempo	5º tiempo	media
5	0,053	0,074	0,065	0,031	0,022	0,049
10	0,113	0,636	0,314	0,061	0,093	0,2434
15	3,897	3,214	1,705	1,089	0,335	2,048
20	10,888	1,203	2,605	6,037	21,124	8,3714
25	326,25	32,661	1907,27	3,209	2,917	454,4614
30	6353,01	16870	1148,44	3657,17	2751,48	6156,02
35	24565	125832	200932	521083	342143	242911

Se puede observar que hasta un tamaño de entrada 20, el tiempo de ejecución del programa es relativamente pequeño (8ms de media). Sin embargo, al seguir aumentando el tamaño del problema (35), ya obtenemos un tiempo de ejecución de 242 segundos de media.

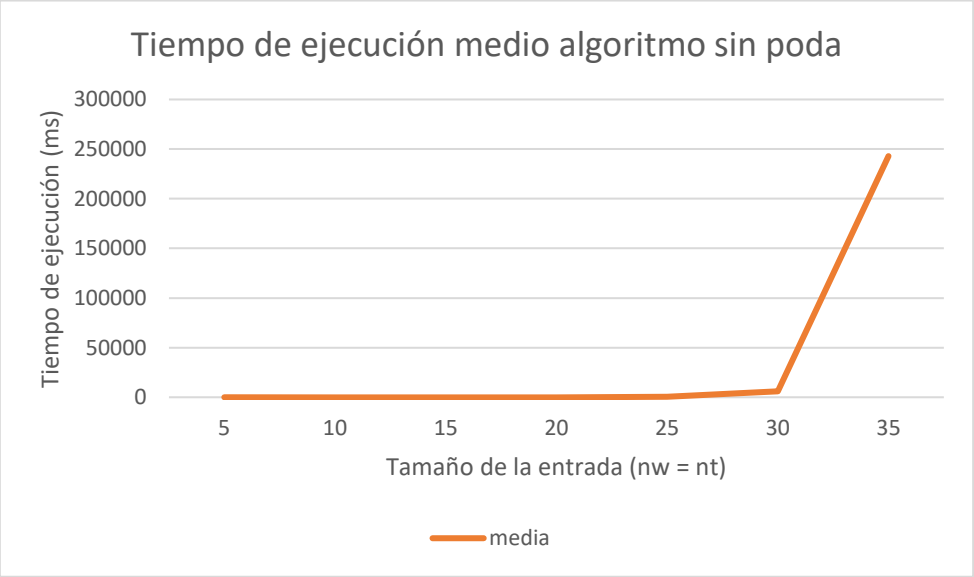
Sin embargo, también queda en evidencia que, para algunos casos, la poda puede ser un gran aliado. Al fijarnos en la fila de tamaño 25, se puede ver que el tiempo de ejecución varía de forma drástica, pudiendo ser de 1907ms (1,9s) en la 3º medición y de 3,2ms y 2,9ms en la 4º y en la 5º. La explicación de esta variación del tiempo de ejecución se debe a la poda. En la 3º medición apenas se han hecho podas, dejando ese tiempo tan alto, mientras que, en los otros dos casos, se ha hecho una gran cantidad de podas, eliminando una gran cantidad de caminos y combinaciones, disminuyendo de forma muy considerable el tiempo de ejecución del caso.

Esto también pasa en la fila de tamaño 35, donde la 1º medición es de 24 segundos, y la 4º es de 521 segundos.

Contraste del Estudio Teórico y Experimental

Los resultados obtenidos permiten afirmar que el algoritmo de backtracking sin poda es prácticamente inviable incluso para instancias moderadas, mientras que la versión con poda logra resolver eficientemente instancias pequeñas y medianas, aunque conserva un crecimiento claramente exponencial.

En la siguiente gráfica se muestra el aumento del tiempo de ejecución en función del tamaño de entrada, en promedio.



A partir del tamaño 30, ya podemos ver una leve subida del tiempo de ejecución, sin embargo, a partir del tamaño 35 es evidente el drástico cambio en el tiempo de ejecución, saltando, en media, de 6 segundos hasta 250, lo que respalda los resultados obtenidos en el análisis teórico, donde se indicaba que el algoritmo tenía orden de $O(nw^{nt})$.

CONCLUSIONES

En conclusión, la realización de está práctica nos ha ofrecido una oportunidad para afianzar y consolidar los conocimientos sobre algoritmos voraces y backtracking. A pesar de la complejidad que ambos enfoques presentan en un primer momento, su implementación práctica favorece a una mejor comprensión y permite adquirir soltura en su aplicación.

Por otra parte, en cuanto a la organización de los contenidos, nos hubiera gustado abordar ambos temas de manera más continua en el tiempo, ya que separar las explicaciones durante varias semanas puede dificultar la retención de conceptos, añadiendo una dificultad extra la hora de enfrentarse a la práctica.

Además, el hecho de tener que desarrollar dos ejercicios completos, aunque el plazo de entrega sea razonable, supone una carga de trabajo considerable, especialmente teniendo en cuenta que coincide con la entrega de otros proyectos de otras asignaturas y se sitúa muy cerca del inicio del periodo de exámenes finales. Por ello, una posible mejora, podría ser dividir la práctica en dos entregas independientes o escalonadas, permitiendo así una mejor gestión del tiempo y una asimilación más progresiva de los contenidos.