Proyecto Biblioteca
GRUPO 17

Integrantes:

Juan Manuel Chanes.

juanmanuel17102001@gmail.com.

Santiago Herrero.

herrerosanty@gmail.com.

Introducción al problema:

Se tiene un conjunto de libros con las siguientes características:

- Título
- Autor
- Puntaje
- Género
- Ejemplares en stock
- Cantidad de hojas

Los cuales se deben repartir de manera tal que dado un máximo de alumnos y un puntaje mínimo para la aprobación se genere un listado de los libros que debería leer cada alumno maximizando la cantidad de alumnos aprobados. Como se pide buscar la mejor solución posible, fue necesaria la implementación de un algoritmo de complejidad exponencial.

A lo largo del informe se verá el modelado del problema con detalles, la implementación del algoritmo con sus respectivas explicaciones, resultados y mejoras del mismo a lo largo del tiempo.

Modelado del problema:

Dado que para este trabajo utilizamos la biblioteca STL en C++, comenzamos planteando una clase secundaria “Libro” la cual aporta algunos métodos que facilitan el manejo de los mismos (controlar la cantidad de ejemplares que se tiene, obtener el título o cualquier otro dato del propio libro). Posteriormente construimos la clase “Inventario” que contenía una lista de elementos tipo Libro que permitía tener los libros en una estructura eficiente con bajo tiempo para la inserción de elementos (O(1)), construimos los métodos públicos básicos para poder agregar elementos y posteriormente las funciones necesarias para resolver el problema. Levantamos el archivo que nos proporcionó la cátedra en el Inventario.

La solución la retornamos como un vector<map<int,Libro>> que cada posición del vector contendrá el listado que debería leer el alumno para poder aprobar, la razón por la cual elegimos un mapa fue para poder comprobar la existencia de un elemento en el conjunto de manera eficiente(O(log(n))). La función que genera esa solución se describe en el siguiente punto.

Implementación algoritmo:

Para la implementación del algoritmo, primero fue necesario identificar los datos que eran relevantes para realizar el track de la mejor solución, dado que el algoritmo tiene una estructura particular. Comenzamos implementando BackTracking sin condiciones y guardando los resultados sobre una lista<Pares> para probar su funcionamiento (agregabamos en una lista el par <Persona, libro> y para considerar la lista como una solución válida no podía contener pares repetidos. Posteriormente, y una vez que la idea ya estaba más formada, nos dimos cuenta de que sería mucho más ordenado y eficiente si podíamos retornar un vector de map<int,Libro> en donde cada posición del vector representa al alumno particular. Esto nos permite buscar repetidos sobre el mapa de cada alumno, también fue posible agregar un valor entero independiente para cada alumno el cual contenía la sumatoria de puntos que tenía hasta el momento, y otro entero que estaba al mismo nivel que el vector que guardaba la cantidad de alumnos aprobados en la solución abaratando los costos.

```
void Inventario::generarConjuntoLibrosParaNChicos(int alumnosMax, int puntajeAprobacion, vector<map<int,Libro>> & retorno)
{
    solucion mejor;
    mejor.sol.resize(alumnosMax);
    for (int i = 0; i < alumnosMax; i++)
        mejor.sol[i].puntajeActual=0;
    mejor.aprobados = 0;
    int estados=0;
    /*int alumnoActual = solucionRapida(mejor,alumnosMax,puntajeAprobacion,estados);  // acotar el conjunto
    solucion temp;
    temp = mejor;
    backTracking(temp,mejor,alumnosMax,puntajeAprobacion,alumnoActual,estados);
    *///Descomentar para usar backtracking y comentar solucionHeuristica
    solucionHeuristica(mejor,alumnosMax,puntajeAprobacion,estados);
    for(int i = 0 ; i < alumnosMax ; i++)
        retorno[i]=mejor.sol[i].datos;
    cout<<estados<<endl;

}
```

Este es el método público que es llamado por fuera de la clase con los parámetros solicitados (alumnosMáximos, puntajeAprobacion, conjunto retorno (vector<map<int,Libro>>), principalmente le pide los datos al usuario y el conjunto donde quiere guardar la mejor solución.

```
int Inventario::solucionRapida(solucion & mejorSolucion, int alumnosMax, int valorEsperado,int & estados)
{
    int personaActual = 0;
    typename list<Libro>::iterator itList = datos.begin();
    while ((itList != datos.end()) && (personaActual < alumnosMax))    // recorro conjunto e inserto listas individuales
    {
        while ((itList->getPuntaje() >= valorEsperado) && (itList->getEjemplares() > 0) && (personaActual < alumnosMax)){
            mejorSolucion.sol[personaActual].datos[itList->getId()]=*itList;  // inserto el libro en el mapa en la id del libro
            mejorSolucion.sol[personaActual].puntajeActual=itList->getPuntaje();
            mejorSolucion.aprobados++;
            itList->restarEjemplar();
            cantLibros--;
            personaActual++;
            estados++;
        }
        itList++;
    }
    return personaActual;  // retorno el alumno actual
}
```

Esta función es una solución que se nos ocurrió para hacer más eficiente la resolución del problema. Nos dimos cuenta de que los libros que tenían un puntaje mayor o igual al puntaje de aprobación pueden ser agregados como elementos sueltos a cada alumno particular en su mapa y serían una solución ideal. **Esto acota mucho el tiempo de ejecución ya que se puede resolver el problema de manera polinómica (en algunos casos) y no exponencial**. Retorna el último alumno que pudo completar.

```
void Inventario::backTracking(solucion & solucionActual, solucion & mejorSolucion, int alumnosMax, int valorEsperado, int alumnoActual,int & estados)
{
    estados++;
    if ((this->vacio()) || (alumnosMax == solucionActual.aprobados)){
        if (solucionActual.aprobados > mejorSolucion.aprobados)
            mejorSolucion=solucionActual;
    }else{
        int persona = alumnoActual;
        while ((persona < alumnosMax) && !this->vacio()){
            Libro agregar = this->tomarPrimero();//tomo el primer libro de la lista
            solucionActual.sol[persona].puntajeActual+=agregar.getPuntaje();//sumo puntaje para acotar complejidad
            // si no agrego libro por demas y el libro no pertenece al conjunto del alumno, lo tomo como solucion valida
            if (((solucionActual.sol[persona].puntajeActual-agregar.getPuntaje()) < valorEsperado)&&(solucionActual.sol[persona].datos.find(agregar.getId()) == solucionActual.sol[persona].datos.end())){
                if (solucionActual.sol[persona].puntajeActual >= valorEsperado)
                    solucionActual.aprobados++;
                solucionActual.sol[persona].datos[agregar.getId()] = agregar;
                backTracking(solucionActual,mejorSolucion,alumnosMax,valorEsperado,alumnoActual,estados); //recursiva
                if (solucionActual.sol[persona].puntajeActual>= valorEsperado)
                    solucionActual.aprobados--;
                solucionActual.sol[persona].datos.erase(agregar.getId());
            }
            this->sumarLibro(agregar.getId());// devuelvo el libro al conjunto
            solucionActual.sol[persona].puntajeActual-=agregar.getPuntaje();
            persona++;
        }
    }
}
```

Función de backtracking con condiciones de poda que en el peor caso comienza desde la persona 1 si solución rápida no puede completar alumnos por sí solo, caso contrario arranca desde la primera persona que la función retorna. Genera todas las soluciones posibles y se queda con la que mayor cantidad de alumnos aprobados tenga, corta cuando detecta que está agregando libros de más a un alumno en particular.

PseudoCodigo backtracking

```
void backtracking(conjunto datos,solucion mejor,solucion actual,int puntaje,int alumnosMax){
    if ((datos esta vacio) o (todos aprobados)){
        if(actual es mejor solucion que mejor){
            mejor=actual;
        }
    }else
        int persona=0;
        while((persona<alumnosMax) y (datos no esta vacio)){
            agregar=tomar primer elemento de datos
            actual.puntaje+=agregar.puntaje
            if((no agrego libros de mas en persona) y (agregar no pertenece al conjunto de persona)){
                if(persona aprobo){
                    actual.aprobados++
                }
                actual.insertalibro(agregar,persona)
                backtracking( datos, mejor, actual, puntaje, alumnosMax)
                if(persona aprobo){
                    actual.aprobados--
                }
                actual.borrarlibro(agregar,persona)
            }
            actual.puntaje-=agregar.puntaje
            datos.devolver(agregar)
            persona++
        }
}
```

Mapa de Exploración:

Dado que la cantidad de soluciones que se pueden generar en backtracking es muy grande, hicimos un ejemplo a pequeña escala para poder explicarlo. El ejemplo consta de 3 personas y 5 libros en donde existen 2 ejemplares del libro N°1

- L1, 10 puntos, 2 ejemplares
- L2, 20 puntos, 1 ejemplar
- L3, 10 puntos, 1 ejemplar
- L4, 10 puntos, 1 ejemplar
- L5, 10 puntos, 1 ejemplar

Criterio de Ramificación:

Se divide en las diferentes personas que se le puede asignar tal libro, teniendo en cuenta que una persona no puede tener dos libros iguales.

Criterio de profundidad:

La profundidad del árbol es determinada por la cantidad de libros o que llegue a aprobar a todos los alumnos.

Estados del Árbol:

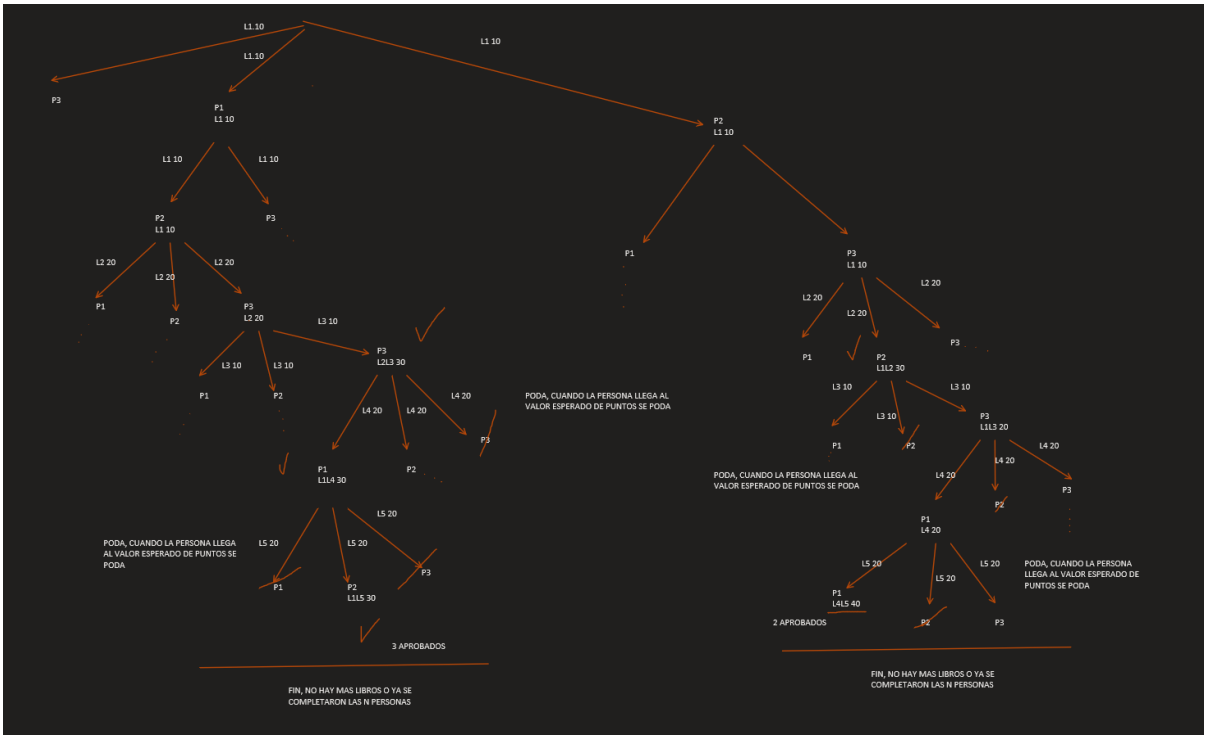
Cada estado representa a la persona que podría tener ese libro, en ellos se tiene como dato los libros que debería leer y el puntaje total de la suma de dichos libros, las restricciones de los siguientes estados son que un estado no puede tener libros repetidos en la “lista” de libros a leer.

Mejor Solución(3 aprobados)

P1 L4L1
P2 L1L5
P3 L2L3

Solución Actual(2 aprobados)

P1 L4L5
P2 L2L1
P3 L1L3



Descripción:

- Las tildes representan: los alumnos que ya llegaron al valor para aprobar
- Los valores tachados representan la poda
- Las líneas horizontales al fondo representan que ya no hay más libros asignables.
- Los puntos suspensivos serían más soluciones pero para un ejemplo práctico no lo realizamos

Si consideramos solamente el backtracking (no consideramos el solucionRapida()).
La solución inicial sería el conjunto solución vacío, dado que de ahí parte, y comienza a generar una primera solución para tomarla por mejor (una vez que se apruebe por lo menos un alumno o el conjunto se quede sin datos). En el caso que dibujamos, la primera mejor solución sería la de la izquierda.

Posteriormente genera una segunda posible solución (rama derecha) la cual se extiende hasta que se vacía el conjunto y se compara con la primera en base a la cantidad de aprobados.

Resultados de la ejecución con y sin poda:

La poda es un recurso que nos permite descartar los casos que no tienen sentido y así se minimiza la cantidad de estados que necesita intentar el backtracking para llegar a la mejor solución.

En este caso, lo que hacemos es que cuando una persona ya posee el puntaje necesario para aprobar no se le agrega más libros, ya que no hace falta. Con esto nos ahorramos pasar por combinaciones que nunca van a ser la mejor solución.

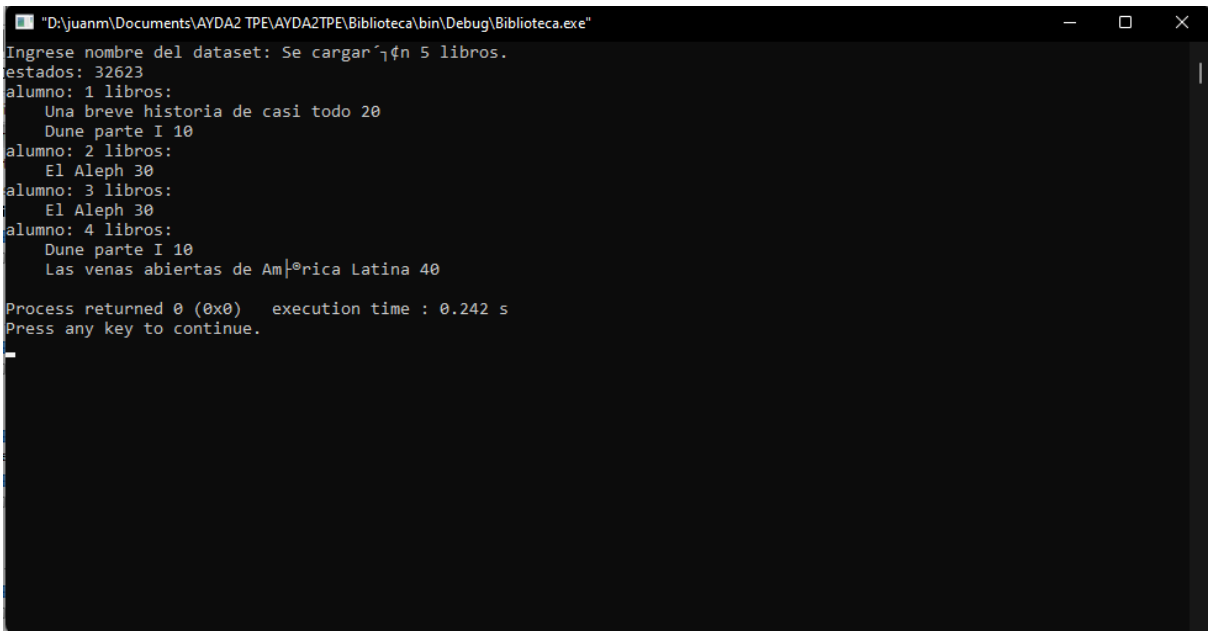
Para poder realizar una comparación, agregamos una variable entera que se sumaba por cada vez que se llama recursivamente al algoritmo.

Descripción:

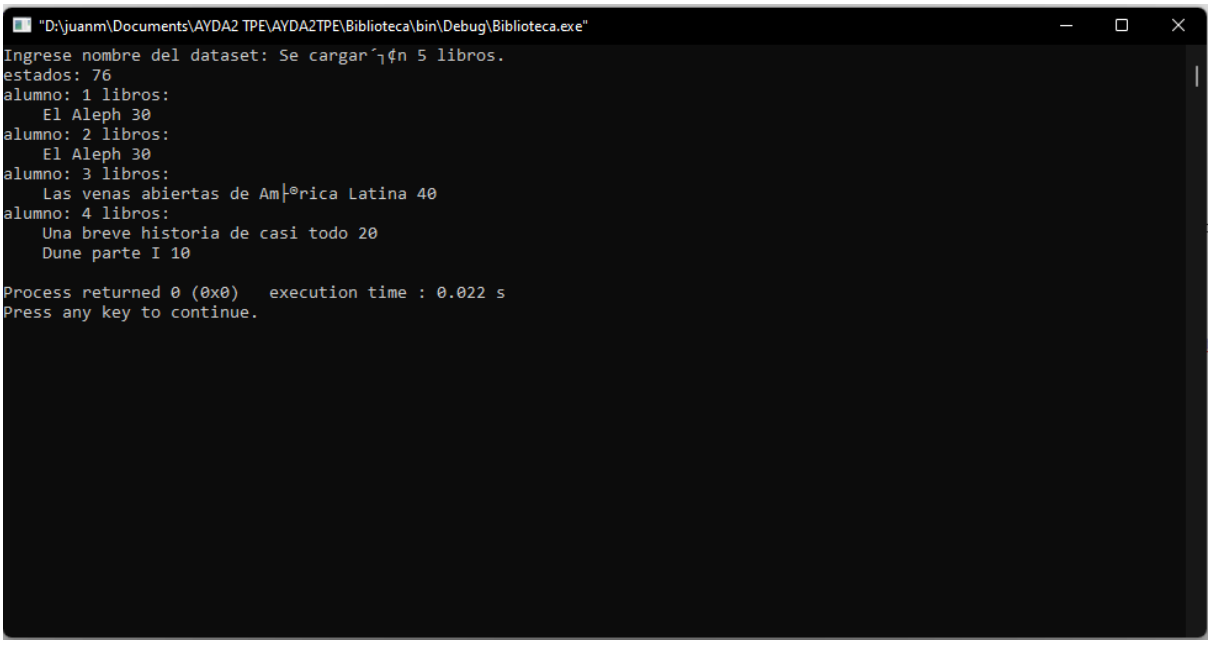
Para la siguiente prueba utilizamos siempre los mismos parámetros

- DataSet 1.
- 6 alumnos como máximo.
- 30 como puntaje de aprobación.

Descripción: Cantidad de estados sin ningún tipo de poda (233.503 estados).



Descripción: Cantidad de estados con condiciones de poda y sin SolucionRapida() (32.623 estados, aproximadamente 7 veces más eficiente con respecto de la solución sin poda).



Descripción: Cantidad de Estados con la función SolucionRapida() y la condición de poda. Cabe aclarar que se ejecutó backTracking dado que el cuarto alumno posee 2 libros y SolucionRapida() no puede hacer eso.
(73 estados, aproximadamente 420 veces más eficiente que backtracking con poda y 2900 veces más rápido que la solución sin poda).

Como podemos ver la cantidad de veces que se llamó a backTracking se fue reduciendo a medida que agregamos condiciones de poda y soluciones

complementarias al problema, siempre obteniendo una solución similar (en cuanto a la distribución de los libros) pero siempre con la misma cantidad de alumnos aprobados (4).

Resultados:

Tabla de pruebas:

Alumnos Maximo	Puntaje Aprobación	DataSet	Estados Con Poda	Estado sin poda
6	30	1	76	88
4	120	2	277889	3346913
5	100	2	13471166	Sin Solución
4	100	3	21857	326369

En esta tabla se ve como la condición de poda disminuye la cantidad de veces que se llama a la función backtracking. También se ve como mejora el rendimiento al punto que puede dar solución en un caso que sin poda no lo hace.

Segunda parte.

Introducción:

Se nos pidió que diseñemos un algoritmo que nos diera una solución aproximada y no la mejor de manera más eficiente debido al aumento de peticiones y de la cantidad de alumnos que se esperaban aprobar dados conjuntos de mayor volumen. Por lo tanto, pudimos disminuir la complejidad temporal del programa a un tiempo polinómico.

Implementación algoritmo heurístico:

Se solicitó la implementación de un algoritmo que proporcionara una solución aproximada a la mejor. Para ello, se nos ocurrieron dos ideas:

- La primer idea que se nos ocurrió, es ordenar el conjunto de mayor a menor y agregar el primer elemento disponible (mayor) y avanzar en el conjunto hasta completar el subconjunto de libros que debe leer el alumno, una vez que el puntaje sea mayor o igual al valor esperado se pasa al siguiente alumno y se arranca desde el inicio del conjunto (esto era barato, ya que no había que comprobar la existencia de un libro en el conjunto y no hay que intentar pasarse del valor esperado con la menor diferencia posible).

- Ordenar el conjunto de mayor a menor y comenzar a recorrerlo desde el principio (mayores). Agregamos el elemento al subconjunto siempre que no nos pasemos del valor esperado. Cuando se pasa del valor, recorre el conjunto hacia adelante (menores) sin agregar los elementos buscando uno que complete el subconjunto o que sume sin pasarse del valor esperado, si aún es posible intentar completarlo (que haya libros disponibles), repite este último ciclo.

Decidimos implementar la segunda opción ya que, a pesar de ser más costosa (es necesario chequear la existencia del elemento en el subconjunto del alumno), pensamos que la solución que va a proporcionar en la mayoría de los casos será mucho mejor a la que podría proporcionar la primera opción (con respecto a la cantidad de alumnos aprobados).

```cpp
void Inventario::solucionHeuristica(solucion & mejorSolucion, int alumnosMax,int valorEsperado,int &estados)
{
    datos.sort();   // ordeno de mayor a menor el conjunto
    int personaActual = 0;
    while (!vacio() && (personaActual < alumnosMax))
    {
        list<Libro>::iterator itLista = datos.begin();
        // tomo libros siempre y cuando no me pase del valor esperado al sumarselo al puntaje acumulado
        while ((itLista != datos.end()) && (mejorSolucion.sol[personaActual].puntajeActual+itLista->getPuntaje() <= valorEsperado)){
            if (itLista->getEjemplares() > 0){  // si no tiene mas ejemplares lo salteo
                mejorSolucion.sol[personaActual].datos[itLista->getId()] = *itLista;
                mejorSolucion.sol[personaActual].puntajeActual+=itLista->getPuntaje();
                itLista->restarEjemplar();
                cantLibros--;
            }
            estados++;
            itLista++;
        }
        // salgo porque lo complete o porque el libro por el que estoy se pasa del puntaje esperado
        while ((itLista != datos.end())&&(mejorSolucion.sol[personaActual].puntajeActual < valorEsperado)){
            list<Libro>::iterator it2 = itLista;
            list<Libro>::iterator ultimoDisponible = datos.end();
            // recorro la lista buscando el libro que complete el conjunto o uno que sume sin pasarse
            while (it2 != datos.end() && ((mejorSolucion.sol[personaActual].puntajeActual+it2->getPuntaje() >= valorEsperado || ultimoDisponible == datos.end()))) {
                if ((it2->getEjemplares() > 0) && (mejorSolucion.sol[personaActual].datos.find(it2->getId()) == mejorSolucion.sol[personaActual].datos.end()))
                    ultimoDisponible = it2;
                it2++;
                estados++;
            }
            if (ultimoDisponible != datos.end()){   // si pude tomar un libro lo agrego
                mejorSolucion.sol[personaActual].datos[ultimoDisponible->getId()] = *ultimoDisponible;
                mejorSolucion.sol[personaActual].puntajeActual+=ultimoDisponible->getPuntaje();
                ultimoDisponible->restarEjemplar();
                cantLibros--;
            } else if (itLista != datos.begin())   // puede ser que desde el punto en el que estoy hacia adelante no hay mas libros, recorro desde el principio y tomo el primero
                itLista = datos.begin();    // doy una ultima vuelta
            else
                itLista = datos.end();  // ya no hay caso, salgo
            estados++;
        }
        // si lo pude aprobar sumo aprobados
        if (mejorSolucion.sol[personaActual].puntajeActual >= valorEsperado)
            mejorSolucion.aprobados++;
        personaActual++;
    }
}
```

Descripción:

El algoritmo heurístico lo que hace es ordenar el conjunto de mayor a menor para posteriormente ir completando los alumnos tomando el primer libro disponible siempre y cuando no se pase del puntaje de aprobación. Una vez que el libro que se intenta agregar supera el puntaje, el algoritmo entra en otro ciclo que avanza en el conjunto hasta encontrar el libro que completa el subconjunto con lo justo o que se pase lo menos posible del puntaje necesario para aprobar.

Resultados:

Alumnos Maximo	Puntaje Aprobación	DataSet	Estados Back	Estados Aproximación	Cantidad Alumnos Aprobados Back	Cantidad Alumnos Aprobados Aproximación
6	30	1	76	41	4	4
4	120	2	277889	48	4	4
3	50	1	553	14	3	2
100	250	3	Sin solución	29987	Sin solución	100
1000	100	3	Sin solución	294900	Sin solución	432

Como podemos ver en el caso marcado, el algoritmo que propusimos no siempre va a brindar la mejor solución, sino que va a proporcionar una solución aceptable.

Por otra parte, podemos ver que el algoritmo es mucho más eficiente que el backtracking, al punto de que es capaz de proporcionar solución con entradas mucho más grandes de las que soportaba el backtracking.

Conclusiones extraídas del trabajo:

Backtracking es una herramienta muy potente, con un costo muy alto y complejo, para poder generar soluciones y quedarse con la mejor de manera algorítmica. Puede ser utilizada en conjunto con iteraciones simples para resolver problemas complejos en caso de que no sea posible hacerlo mediante funciones de tiempo polinómico.

El algoritmo Heurístico es una alternativa que genera una solución aproximada y no la mejor, logrando que el programa sea mucho más eficaz. Es por ello que es una buena alternativa para no hacer backtracking si se desea que el programa sea más eficiente pero menos preciso.

En resumen, podemos decir que si nuestra necesidad no es tener la mejor solución, nos conviene utilizar el algoritmo heurístico para ganar en eficiencia, pero si se necesita la mejor solución (mayor precisión) deberíamos hacer uso de backtracking aunque tengamos un costo muy alto.