PROGRAMACIÓN 3

TP5 - Backtracking

Ejercicio entregable

Alumno: Pablo Santa María, LU 248940

Introducción

Se solicita diseñar un programa que dé la solución óptima al siguiente problema:

Distribuir familias en una cantidad de días consecutivos para asistir a un evento. Las familias eligen días en orden de preferencia y cada día tiene un cupo limitado de personas. Si la familia no se asigna a su día preferido, le corresponde un bono compensatorio. A mayor cantidad de integrantes y día menos preferido, mayor es el monto del bono.

La solución óptima está dada si se distribuyen las familias de tal manera que *todas* asistan al evento en *algún día de su preferencia,* y el costo total de los bonos a pagar sea el *mínimo posible*.

Enfoque

Para saber si el problema tiene al menos una solución, o (si tiene varias) encontrar la solución óptima, se generan *todas* las combinaciones que sean posibles de familias y el día al que se asigna cada una. Una vez que se tienen todas estas posibilidades, se comprueba que haya una o más soluciones válidas (que no queden familias sin asignar y que los días no superen su cupo máximo) y cuál es la óptima.

Para lograr esto, se utiliza la técnica de Backtracking, primero diseñando un algoritmo recursivo que recorra todo el árbol de exploración llegando a cada uno de los estados finales, el que luego se optimiza incorporando las restricciones y podas del problema en cuestión.

Algoritmo

Se diseñó el algoritmo que recorre el espacio de búsqueda de soluciones, comenzando por una de las familias. Si se van a explorar *todos* los posibles estados de la búsqueda, el orden de éstas no es importante ya que es necesario probar todas las combinaciones posibles, y la solución aparecería sin importar por cuál se comenzó a explorar.

Las familias se asignaron a los días dentro de su lista de preferencia sin tener en cuenta restricciones (fuerza bruta) para obtener el total de combinaciones, sean válidas o no.

Pseudocódigo

```
asignar(familias) {
      if (familias == vacío)
            marcar solución válida/inválida
            actualizar costo
      else
            familia = familias.extraer(familia)
            for (día preferido de familia)
                  agregar(familia, día)
                  asignar(familias)
                  borrar(familia, día)
            familias.agregar(familia)
}
```

Árbol de exploración

Aplicando el algoritmo anterior se generó un árbol de exploración que se recorre en profundidad (depth first search). Para ilustrar cada nodo del árbol, se simplificó la entrada de datos, suponiendo que se tienen 3 familias (A, B, C) para asignar y que cada una elige 3 días (1, 2, 3) para asistir.

Estado inicial

En su estado inicial, el árbol se encuentra vacío y el arreglo de familias totalmente lleno, indicando que no existen familias asignadas a algún día del evento.

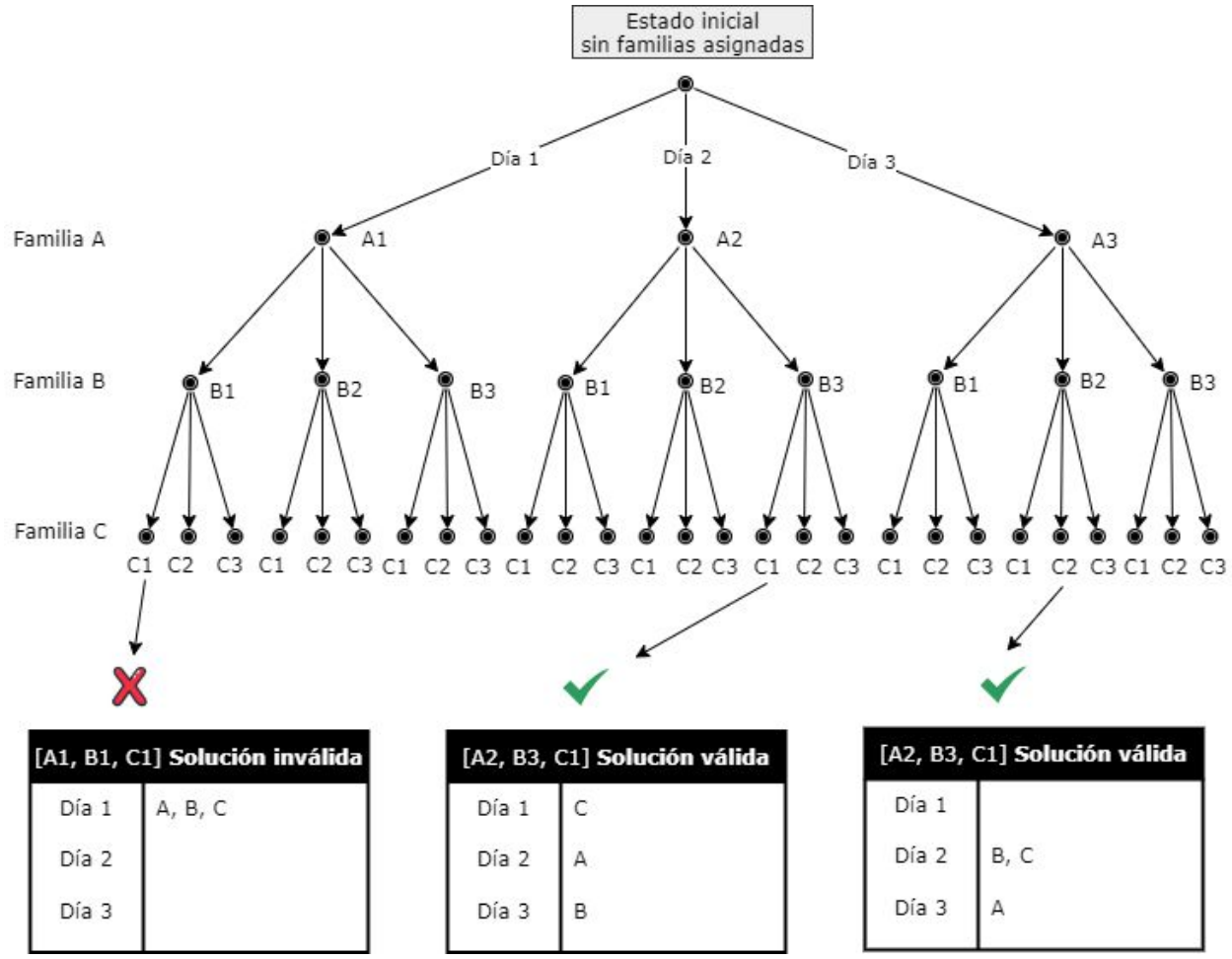
Criterio de ramificación

El árbol de búsqueda se va ramificando a medida que se asigna una familia a un día dentro de sus elegidos. Así, el primer nodo se designa como **A1**, indicando que a la familia A se le otorga el día 1 en su lista de favoritos. El segundo nodo **B1**, corresponde a la familia B en el día 1 de su lista, y el tercero es **C1**. Esto sucede con cada uno de los días de la lista de cada familia. Por ello, cada llamado recursivo del algoritmo se ramifica en tantos días preferidos existan para cada una (en este caso 3).

Nótese que el día 1, 2 o 3 corresponden a la posición del día elegido por la familia, y no a un día específico del evento. Es posible que el primer día de elección de una familia sea distinto del primero elegido por otra.

Profundidad del árbol

En cada nivel del árbol se encuentra una familia con una serie de combinaciones de su lista de días. La profundidad del árbol se corresponde con la cantidad de familias que existen para asignar. Se obtiene así, por cada una de las ramas, una de las secuencias de designación de las familias a un día específico.



[A1, B1, C1] **Solución inválida**	
Día 1	A, B, C
Día 2	
Día 3	

[A2, B3, C1] **Solución válida**	
Día 1	C
Día 2	A
Día 3	B

[A2, B3, C1] **Solución válida**	
Día 1	
Día 2	B, C
Día 3	A

Estados

En cada llamado del algoritmo recursivo se obtienen distintos estados, los cuales pueden ser:

- Inicial
    - Quedan todas las familias por asignar.
- Parciales
    - Estados en los cuales quedan familias por asignar. Corresponde al momento en que un nodo interno del árbol no ha generado todos sus hijos. Para contar estos estados se realiza la sumatoria de *días preferidos ^ familias asignadas hasta ese nivel*. En este caso es 1 + 3 + 9 + 27 = 40.

- Finales
    - Todas las familias están asignadas a algún día. Corresponde a un nodo hoja del árbol donde no es posible avanzar. El camino desde la raíz hasta el nodo hoja puede ser solución o no, y esto depende de las restricciones del problema. La cantidad de estos estados es *días preferidos ^ familias.* En este caso es 3 ^ 3 = 27.
- Soluciones
    - Es un estado final que cumple con las restricciones del problema. Puede ser la solución óptima o no. En este estado todas las familias están asignadas a un día que tiene vacantes de su preferencia, y es óptimo si el monto es el menor obtenido.

Datos almacenados

Se hizo uso de estructuras de datos y variables que el algoritmo utiliza como recursos compartidos en cada llamado:

En cada estado existe un arreglo dinámico de familias donde están aquellas que quedan por asignar y el monto de los bonos que se van entregando.

Globalmente se mantiene una lista de días, los cuales se van poblando con las familias asignadas. Esta lista se utiliza para para generar un mapeo entre los días del evento y las familias que se han distribuído en cada uno, a modo de cronograma. Este sirve para presentar una solución obtenida.

Restricciones

Las restricciones de generación de estados del problema son:

- La familia tiene que estar asignada a un día *dentro de su lista* de preferencia.
- No se puede asignar a una familia a *más de un día.*
- No se puede asignar a una familia a un día que no tenga vacantes disponibles.

El algoritmo diseñado, restringe implícitamente las dos primeras, ya que en cada llamado trata de incorporar a la familia sólo en los días preferidos y no conoce los demás. Si esto no es posible, no intenta asignarla en otro lugar, simplemente esa familia queda afuera.

Para no generar estados que violen la tercer restricción, se modificó el algoritmo para que no haga un llamado recursivo si no hay vacantes en ese día:

```
for (día preferido de familia)
    if (dia.vacantes >= familia.miembros)
        agregar(familia, día)
        asignar(familias)
        borrar(familia, día)
```

Registro de resultados

Se utilizaron dos datasets proveídos por la cátedra:

- *familias-1*, el cual contiene **20** familias con una lista de 3 días preferidos.
- *familias-2*, el cual contiene **60** familias con una lista de 3 días preferidos.

El evento está disponible durante **10** días consecutivos y cada uno tiene una capacidad de **30** personas.

Se incorporaron al algoritmo variables que llevan la cuenta de los estados que se van generando. Se cuentan los estados parciales y finales, teniendo en cuenta la restricción de vacantes por día. Esto es: el algoritmo no se llama recursivamente si la familia no va a entrar en ese día, por lo tanto no se aumentan los contadores de estados.

También se utilizó una variable local al método que lleva la suma del costo de bonos que se van entregando si la familia no es asignada a su primer día de preferencia. Al finalizar el recorrido por todos los estados solución, se registró el menor monto total obtenido.

Se corrió el algoritmo arrojando los siguientes resultados:

Dataset familias-1

Estados parciales: `2.205.413`

Estados finales: `633.533`

Monto mínimo: `$625`

Dataset familias-2

Sin datos

No fue posible obtener resultados, ya que el tamaño de entrada de los datos aumenta en un 200%, haciendo que el algoritmo no pueda terminar en un tiempo razonable.

Poda y optimización

Para reducir el tiempo de ejecución, es necesario realizar una poda al árbol del espacio de búsqueda. Esta consiste en no avanzar por una rama si se descubre que ese camino no va a dar una solución válida, o la solución óptima. Cuando sucede esto, la búsqueda por esta rama se interrumpe y se retrocede para continuar por otra.

El criterio de poda que se implementó, corta la ejecución en esa rama cuando:

- No hay vacantes suficientes para asignar a una familia en el día correspondiente a ese estado.

- El monto sumado hasta el momento es mayor o igual al monto total obtenido en otra solución válida.

Estas dos condiciones de poda se ubicaron en el algoritmo justo antes de agregar a la familia al siguiente día.

```
if (dia.vacantes >= familia.miembros && costoParcial <= costoTotal)
                agregar(familia, día)
                asignar(familias)
                borrar(familia, día)
```

Por otro lado, se ordenó la lista de familias a asignar por cantidad de miembros, en forma descendente. Esto podría tener un gran impacto en la ejecución, ya que la poda por no quedar vacantes para las familias numerosas se realizaría al principio de la exploración y el logaritmo evita avanzar demasiado en la rama; por lo tanto se hace menos retroceso.

Se corrió el algoritmo optimizado arrojando los siguientes resultados:

Dataset familias-1

Estados parciales: 1.680

Estados finales: 5

Monto mínimo: $625

Dataset familias-2

Estados parciales: 5.353.295

Estados finales: 7.585

Monto mínimo: $260