Universidad de Granada

PRÁCTICA 5

ALGORITMOS PARA EXPLORACIÓN DE GRAFOS

Antonio Jiménez Rodríguez 2D (SubGrupo D2)

Problema asignado

Ejercicio 9:

Dado un conjunto de n enteros, necesitamos decidir si puede ser descompuesto en dos subconjuntos disjuntos cuyos elementos sumen la misma cantidad. Resolver el problema mediante una técnica de exploración en grafos.

1. Análisis y notación a utilizar

Como nos dice el enunciado, queremos comprobar si a partir de un conjunto dado de números enteros podemos dividirlo en dos subconjuntos, donde la suma de los elementos que se encuentren en cada subconjunto sea la misma en los dos.

Sabiendo esto vamos a usar la siguiente notación para la explicación de nuestro problema:

- Llamamos **n** al número de elementos del conjunto donde n $\geq$ 2.
- Denotamos al conjunto de n enteros dados como $X = \{ x_1, x_2, ..., x_n \}$.
- LLamamos t_1 al tamaño del primer subconjunto y t_2 al tamaño del segundo subconjunto.
- Denotamos al primer subconjunto que formará la solución como $A = \{ a_1, a_2, ..., a_{t_1} \}$ y al segundo como $B = \{ b_1, b_2, ..., b_{t_2} \}$.
- Llamamos ΣA a la suma de todos los elementos del subconjunto A.
- Llamamos ΣB a la suma de todos los elementos del subconjunto B.

  Los elementos del conjunto no pueden ser todos igual a cero.

Vamos a resolver el problema propuesto haciendo uso de la técnica de backtracking en la que generaremos el árbol de soluciones del problema y lo recorreremos para hallarlas.

2. Diseño de componentes

- **¿Qué representación usamos para las soluciones del problema?**

Como hemos visto en las notaciones vamos a usar dos subconjuntos A y B. Estos serán dos vectores que contendrán los valores de enteros del primer conjunto correspondientes para que se cumpla la solución del problema.

- **¿Qué restricciones implícitas encontramos en el problema?**

Son los valores que cada x_i puede tomar para construir la solución. En nuestro caso esto lo vamos a representar como: estar en el conjunto uno o estar en el conjunto dos, por lo que para cada nodo tendremos 2 hijos únicamente.

- **¿Qué restricciones explícitas encontramos en el problema?**

Las restricciones externas al proceso de encontrar la solución sería tener que haber recorrido todos los elementos del conjunto dado. No puede quedar ningún elemento fuera de los dos subconjuntos solución. Esto significa que de existir una posible solución solo la encontraremos en el último nivel.

- **¿Qué estructura de árbol/grafo implícito vamos a utilizar?**

Vamos a utilizar una estructura de árbol binario donde cada nivel i del árbol se corresponderá con el elemento i-ésimo del conjunto X. Entenderemos que al escoger el hijo de la izquierda del elemento i-ésimo como parte de la solución, ese elemento x_i será parte del subconjunto A. Por el contrario si elegimos el hijo derecha x_i será parte del subconjunto B.

- **¿Cuál es nuestra función objetivo?**

Nuestro criterio de parada será encontrar todas las soluciones del problema, por lo tanto será necesario haber generado todo el árbol de soluciones para el conjunto X. La solución estará en los nodos cuyo valor sea 0, ya que lo que haremos será restar las cantidades de cada subconjunto generado para la rama en la que nos encontremos.

- **¿Cuál es la función de poda?**

Nuestra función de poda va a cambiar dependiendo de la decisión, añadirlo al conjunto A, que se corresponde con la rama izquierda o añadirlo al conjunto B que se corresponde con la rama derecha de nuestro árbol. Siendo $\{X \setminus A \setminus B\}$ el conjunto de elementos que quedan por explorar en el conjunto X, para la rama izquierda **no** será factible explorarla si ΣA es menor que ΣB y la $\Sigma A + \Sigma\{X \setminus A \setminus B\}$ y en $\{X \setminus A \setminus B\}$ sólo hay valores positivos.

2.1. Algoritmo diseñado

Nuestro objetivo es construir 2 subconjuntos de números enteros que serán la solución de nuestro problema. El diseño sería el siguiente:

- Se irá generando el árbol binario completo del conjunto X, donde cada nivel sería tomar de decisión de cada elemento del conjunto para añadirlo en un subconjunto o en otro.
- En el nodo raíz vamos a tener el elemento x_2 ya que al elemento x_1 lo colocaremos siempre en el conjunto A y a partir de ahí empezaremos a construir la solución.
- Este árbol lo recorreremos en profundidad de derecha a izquierda y las decisiones las iremos guardando en un vector llamado decisiones cuyo índice corresponderá con el de X. Tambíen en cada paso estarán los correspondientes valores en cada subconjunto.
- Al llegar a un nodo hoja se comprobará si este es solución. Si es así los valores actuales del subconjunto A y el subconjunto B se guardarán el conjunto de soluciones posible

```
PROCEDIMIENTO BACKTRACKING Subconjuntos(X[0..n], Decisiones[0..n], A[0..t1], B[0..t2])
i = 0       //Primer elemento del conjunto X
A U {X[i]}
Decisiones[i] = 0
i = i + 1
Decisiones[i] = 2    //Inicializamos al valor nulo
MIENTRAS QUE i >= 0 HACER
    Decisiones[i] = Decisiones[i] - 1
    SI Decisiones[i] == -1 ENTONCES
        A \ {X[i]}
        i = i - 1      //Generados todos los descendientes de i, backtraking
    SI NO
        SI Decisiones[i] == 1 ENTONCES
            SI EsFactible(i, X, Decisiones, A ,B) ENTONCES
                B U {X[i]}
                SI i == n-1 ENTONCES
                    Soluciones = EsSolucion(i, X, Decisiones, A, B)
                SI NO
                    i = i + 1
                    Decisiones[i] = 2
                FIN SI NO
            FIN SI
        FIN SI
        SI Decisiones[i] == 0 ENTONCES
            B \ {X[i]}
            SI EsFactible(i, X, Decisiones, A ,B) ENTONCES
                A U {X[i]}
                SI i == n-1 ENTONCES
                    Soluciones = EsSolucion(i, X, Decisiones, A, B)
                SI NO
                    i = i + 1
                    Decisiones[i] = 2
                FIN SI NO
            FIN SI
        FIN SI
    FIN SI NO
FIN MIENTRAS QUE
DEVOLVER Soluciones
```

Cada posición de decisiones se irá inicializando a 2 como valor nulo y se irá decrementando su valor para estudiar las distintas posibilidades. El valor 1 corresponde con que X[i] será añadido al subconjunto B, con el valor 0 X[i] será añadido al subconjunto A y para el valor -1 significa que ya hemos estudiado todas las posibilidades por lo que realizamos el backtracking.

Dentro de las dos decisiones 0 y 1 tenemos que comprobar que la exploración de esa rama es factible. Para ello usamos la función **EsFactible(...)**. No tenemos una forma clara de ver si una rama es factible o no, ya que si en los elementos restantes por explorar ($\{X \setminus A \setminus B\}$) hay valores positivos y negativos se podría dar el caso de que llegara a una solución. En el caso de que solo hubiera elementos mayores que 0 sin embargo, si vamos a explorar la rama derecha (añadir el elemento en el subconjunto B), siendo ΣB menor que ΣA y la $\Sigma B + \Sigma\{X \setminus A \setminus B\}$ sigue siendo menor que ΣA nunca podremos llegar a una solución para el problema por esa rama. Por lo que no sería factible explorarla. Al igual pasaría con la rama de la izquierda y siendo ΣA menor que ΣB y la $\Sigma A + \Sigma\{X \setminus A \setminus B\}$ menor que ΣB.

```
41  FUNCION EsFactible(i, X[0..n], A[0..t1], B[0..t2], decision)
42  sumatoriaA = 0
43  sumatoriaB = 0
44  sumatoriaRestoX = 0
45  factible = false
46  PARA k=i HASTA n HACER
47      sumatoriaRestoX  = sumatoriaRestoX + X[k]
48      SI X[k] < 0 ENTONCES
49          factible = true
50      FIN SI
51  FIN PARA
52  SI !factible ENTONCES
53      PARA k=0 HASTA t1 HACER
54          sumatoriaA  = sumatoriaA + A[k]
55      FIN PARA
56      PARA k=0 HASTA t2 HACER
57          sumatoriaB  = sumatoriaB + B[k]
58      FIN PARA
59      SI decision == 0 ENTONCES
60          SI sumatoriaA+sumatoriaRestoX >= sumatoriaB ENTONCES
61              factible = true
62          FIN SI
63      SI NO
64          SI sumatoriaB+sumatoriaRestoX >= sumatoriaA ENTONCES
65              factible = false
66          FIN SI
67      FIN SI
68  FIN SI
69  DEVOLVER factible
```

La función **EsSolucion(...)** comprobará si el nodo hoja al que hemos llegado en la ejecución del algoritmo es solución. Si $\Sigma A - \Sigma B = 0$ se guardarán los dos subconjuntos A y B como una de las posibles soluciones de nuestro problema. Si los conjuntos son iguales pero con distinto orden en los elementos no se considera una nueva solución.

3. Cálculo de la eficiencia

```
void Subconjuntos(int *X, int *decisiones, int *A, int *B, int n, int &tamA, int &tamB, solucion* soluciones, int &numSoluciones){
  int i = 0;              // O(1)
  A[tamA] = X[i];         // O(1)
  tamA++;                 // O(1)
  decisiones[i] = 0;      // O(1)
  i++;                    // O(1)
  decisiones[i] = 2;      // O(1)
  while( i >= 0 ){      // O( g(n)+h(n)*(g(n)+f(n)) ) = O( 1 + 2^n*(n+1)) = O(n*2^n)
    decisiones[i] -= 1;   // O(1)
    //¿Ha explorado ya todos los descencientes del nodo?
    if( decisiones[i] == -1 ){     // max { O(n), O(1) }
      tamA--;             // O(1)
      i--;                // O(1)
    }else{
      if( decisiones[i] == 1 ){     // max { O(n) }
        //¿Es factible explorar la rama?
        if( EsFactible(i, X, A, B, n, tamA, tamB, decisiones[i]) ){    // max { O(n), O(n), O(1) }
          B[tamB] = X[i];  // O(1)
          tamB++;          // O(1)
          //¿Ha llegado al nodo hoja?
          if( i == n-1 ){         // max { O(1), O(n), O(1) }
            EsSolucion(A, B, tamA, tamB, soluciones, numSoluciones);   // O(n)
          }else{
            i++;                  // O(1)
            decisiones[i] = 2;    // O(1)
          }
        }else{
          tamB++;          // O(1)
        }
      }
      if( decisiones[i] == 0 ){   // max { O(1), O(n) }
        tamB--;                   // O(1)
        //¿Es factible explorar la rama?
        if( EsFactible(i, X, A, B, n, tamA, tamB, decisiones[i]) ){     // max { O(n), O(n), O(1) }
          A[tamA] = X[i];  // O(1)
          tamA++;
          //¿Ha llegado al nodo hoja?
          if( i == n-1 ){         // max { O(1), O(n), O(1) }
            EsSolucion(A, B, tamA, tamB, soluciones, numSoluciones);    // O(n)
          }else{
            i++;                  // O(1)
            decisiones[i] = 2;    // O(1)
          }
        }else{
          tamA++;                 // O(1)
        }
      }
    }
  }
}
```

En la imagen anterior podemos encontrar el cálculo de la eficiencia linea a linea. En nuestro algoritmo backtracking tenemos un bucle **while** principal donde vamos generando el árbol de decisiones. Dentro de este bucle la eficiencia del cuerpo es de O(n), la cual viene dada a partir de las funciones **EsFactible(...)** y **EsSolucion(...)**. Las demás operaciones que aparecen en este bucle son de un orden constante.

```
57  void EsSolucion(int *A, int *B, int tamA, int tamB, solucion* soluciones, int &numSoluciones){
58    int sumatoriaA = 0, sumatoriaB = 0;      // O(1)
59    solucion sol;                            // O(1)
60
61    for(int i = 0; i < tamA; i++)            // O( i(tamA)+g(tamA) + h(tamA)*(g(tamA)+f(tamA)+a(tamA)) ) = O( 1+1 + tamA (1+1+1) ) = O(tamA)
62      sumatoriaA += A[i];                    // O(1)
63    for(int i = 0; i < tamB; i++)            // O( i(tamB)+g(tamB) + h(tamB)*(g(tamB)+f(tamB)+a(tamB)) ) = O( 1+1 + tamB (1+1+1) ) = O(tamB)
64      sumatoriaB += B[i];                    // O(1)
65                                             // tamA + tamB = n ya que estamos en el nodo hoja -> O(n)
66    if((sumatoriaA-sumatoriaB) == 0){        // max{ O(1), O(n) }
67      sol.A = new int[tamA];                 // O(1)
68      sol.B = new int[tamB];                 // O(1)
69      sol.tamA = tamA;                       // O(1)
70      sol.tamB = tamB;                       // O(1)
71
72      for(int i = 0; i < tamA; i++)          // O( i(tamA)+g(tamA) + h(tamA)*(g(tamA)+f(tamA)+a(tamA)) ) = O( 1+1 + tamA (1+1+1) ) = O(tamA)
73        sol.A[i] = A[i];                     // O(1)
74      for(int i = 0; i < tamB; i++)          // O( i(tamB)+g(tamB) + h(tamB)*(g(tamB)+f(tamB)+a(tamB)) ) = O( 1+1 + tamB (1+1+1) ) = O(tamB)
75        sol.B[i] = B[i];                     // O(1)
76                                             // tamA + tamB = n ya que estamos en el nodo hoja -> O(n)
77      soluciones[numSoluciones] = sol;       // O(1)
78      numSoluciones++;                       // O(1)
79    }
80  }
```

```
55  bool EsFactible(int k, int *X, int *A, int *B, int n, int tamA, int tamB, int decision){
56    int sumatoriaA = 0, sumatoriaB = 0, sumatoriaRestoX = 0;   // O(1)
57    bool factible = false;                                     // O(1)
58
59    for(int i = k; i < n; i++){
60      sumatoriaRestoX = + sumatoriaRestoX + X[i];              // O(1)
61      if(X[i] < 0)                                             // max{ O(1), O(1) }
62        factible = true;                                       // O(1)
63    }
64
65    if(!factible){                           // max{ O(1), O(n) }
66      for(int i = 0; i < tamA; i++)          // O( i(tamA)+g(tamA) + h(tamA)*(g(tamA)+f(tamA)+a(tamA)) ) = O( 1+1 + tamA (1+1+1) ) = O(tamA)
67        sumatoriaA += A[i];                  // O(1)
68      for(int i = 0; i < tamB; i++)          // O( i(tamB)+g(tamB) + h(tamB)*(g(tamB)+f(tamB)+a(tamB)) ) = O( 1+1 + tamB (1+1+1) ) = O(tamB)
69        sumatoriaB += B[i];                  // O(1)
70                                             // tamA + tamB = n ya que estamos en el nodo hoja -> O(n)
71      if(decision == 0){                                       // max{ O(1), O(1), O(1) }
72        if((sumatoriaA+sumatoriaRestoX) >= sumatoriaB)         // max{ O(1), O(1) }
73            factible = true;                                   // O(1)
74      }else{
75        if((sumatoriaB+sumatoriaRestoX) >= sumatoriaA)         // max{ O(1), O(1) }
76            factible = true;                                   // O(1)
77      }
78    }
79
80    return factible;          // O(1)
81  }
```

El número de veces que se ejecute el bucle while depende del peor y el mejor caso. En el mejor caso se ejecutaría n veces, por ejemplo si el primer elemento es mayor que la suma del resto, la única rama que se exploraría el camino de la izquierda, es decir, donde las decisiones son meter los elementos únicamente en el conjunto A. Por lo tanto en el mejor caso la eficiencia de este algoritmo sería de $\Omega(n^2)$. Sin embargo en el peor caso habría que explorar el árbol concreto, por ejemplo que el último elemento de X sea menor que 0, en este caso el algoritmo se ejecutaría el tamaño del árbol, $2^n - 1$. Por lo tanto el tiempo de ejecución del algoritmo en el peor caso es de $O(n2^n - n)$.

4. Detalles de implementación

El lenguaje de programación utilizado para resolver la práctica ha sido C++. Se han tomado las siguientes decisiones de implementación:

- El índice para el conjunto y los subconjuntos va desde 0 hasta tamaño - 1, por simplicidad para el manejo del lenguaje C++.
- La memoria para los conjuntos se hace con un vector de memoria dinámica de tipo entero.
- Se ha creado una estructura solución para guardar cada solución de manera sencilla. La memoria para todas las soluciones se ha realizado con un vector de memoria dinámica de tipo **solucion**.
- Los valores de entrada de X tienen ningún requisito previo.
- Para que A y B sean soluciones es necesario que la suma de los tamaños de cada uno sea igual al tamaño del conjunto X.
- Como mínimo habrá dos elementos en el conjunto X.

Toda la práctica se ha implementado bajo un único fichero problema9.cpp que puede compilarse y ejecutarse con la orden **make**, gracias al makefile implementado. También se puede ejecutar como **./bin/problema9 ...** con los datos deseados.

Algoritmo main

1. Comprueba los argumentos de entrada, tenemos 3 casos:
   - **1.1. ./bin/problema9 ejemplo** que ejecuta el algoritmo con los datos del ejemplo de esta documentación.
   - **1.2. ./bin/problema9 aleatorio** que ejecuta el algoritmo con datos aleatorios
   - **1.3. ./bin/problema9 n X1 X2 ... Xn**
2. Reservar memoria dinámica para X, A, B y soluciones.
3. Introducir los datos de X. Dependiendo del tipo de ejecución se hará de forma aleatoria, uno a uno o por leyéndolos por pantalla.
4. Aplicar el algoritmo Subconjuntos(int *X, int *decisiones, int *A, int *B, int n, int &tamA, int &tamB, solucion* soluciones, int &numSoluciones)
5. Mostrar todas las soluciones del problema por salida estándar.
6. Liberar memoria dinámica.

El algoritmo Subconjuntos(...) se ha implementado en una función fuera del main para tener recogido el código del mismo.

Se hace hincapié en que no se han realizado comprobaciones de errores de rango de valores, asumiendo como prerrequisitos de funcionamiento los siguientes:

- n, t1, t2 deben de ser enteros positivos.
- No se deben introducir para X mas valores del tamaño indicado.

5. Ejemplo

Vamos a ver el funcionamiento del algoritmo con las siguientes entradas:

- n=4
- X = {2, -1, 5, 2}

Con estos valores obtenemos los siguientes resultados:

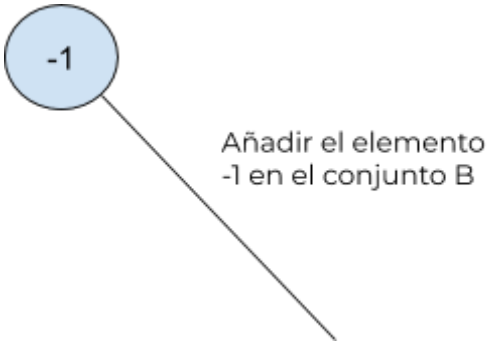
```
antonio@antonio-Lenovo-Y520-15IKBN:~/Documentos/UNIVERSIDAD/5o Año/SEGUNDO CUATRIMESTRE/ALG/PRACTICA5/Pra
tica5_Antonio_Jimenez_Rodriguez$ make
g++ -Wall -g -c -o obj/problema9.o src/problema9.cpp -Iinclude
g++ -o bin/problema9 obj/problema9.o -Iinclude
#bin/problema9 aleatorio
bin/problema9 ejemplo
<<<<<<<<< Ejecucion del Problema 9 con los datos del ejemplo de la documentacion >>>>>>>>>

Conjunto X = {2, -1, 5, 2}

Solucion 1:
Conjunto A = {2, 2}
Conjunto B = {-1, 5}

Hay 1 soluciones.
```

Vamos a ver gráficamente cómo el algoritmo genera el árbol de decisiones del problema.

En la situación inicial el primer elemento lo metemos en el conjunto A y a partir de ahí construimos las posibles soluciones:

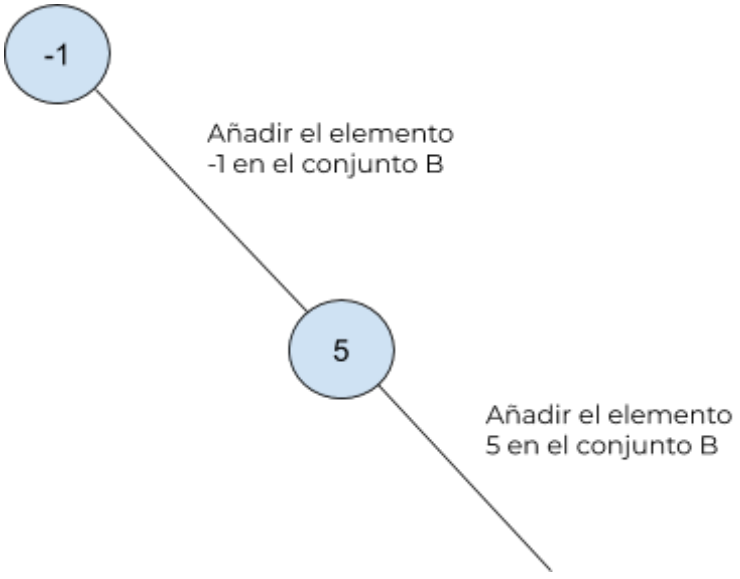
i = 0
X = {2, -1, 5, 2}
A = {2, // añadimos al subconjunto A X[0]
B = { // subconjunto B vacio
i = 1 // valor de i incrementado tras añadir elemento
decisiones = {0, 2} // decisiones[0] = 0 ya que lo elegimos nosotros

Al entrar al bucle while decrementamos decisiones[i], por lo que su valor será 1, es decir X[i] lo añadimos al conjunto B ya que es factible.



i = 1
A = {2,
B = {-1 // añadimos al subconjunto A X[i]
i = 2 // valor de i incrementado tras añadir elemento
decisiones = {0, 1, 2} // inicializamos decisiones[i]

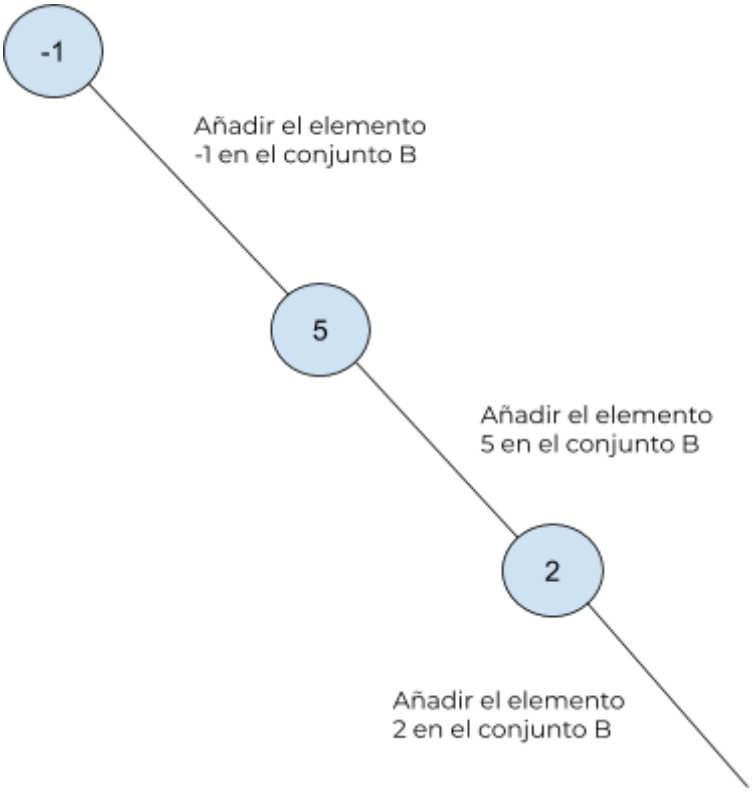
Al volver al inicio del bucle se decrementa decisiones[i] por lo que quedaría decisiones = {0, 1, 1}



i = 2
A = {2,

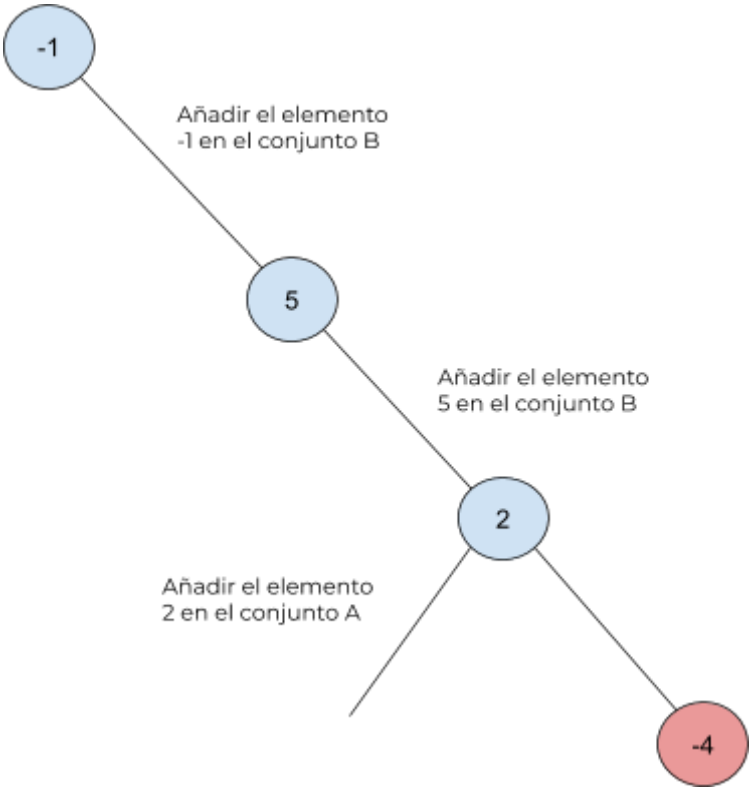
B = {-1, 5 // añadimos al subconjunto A X[i]
i = 3 // valor de i incrementado tras añadir elemento
decisiones = {0, 1, 1, 2} // inicializamos decisiones[i

Al volver al inicio del bucle se decrementa decisiones[i] por lo que quedaría decisiones = {0, 1, 1, 1}



i = 3
A = {2,
B = {-1, 5 // añadimos al subconjunto A X[i]
i = 3 // no incrementamos i porque es hoja
decisiones = {0, 1, 1, 1}

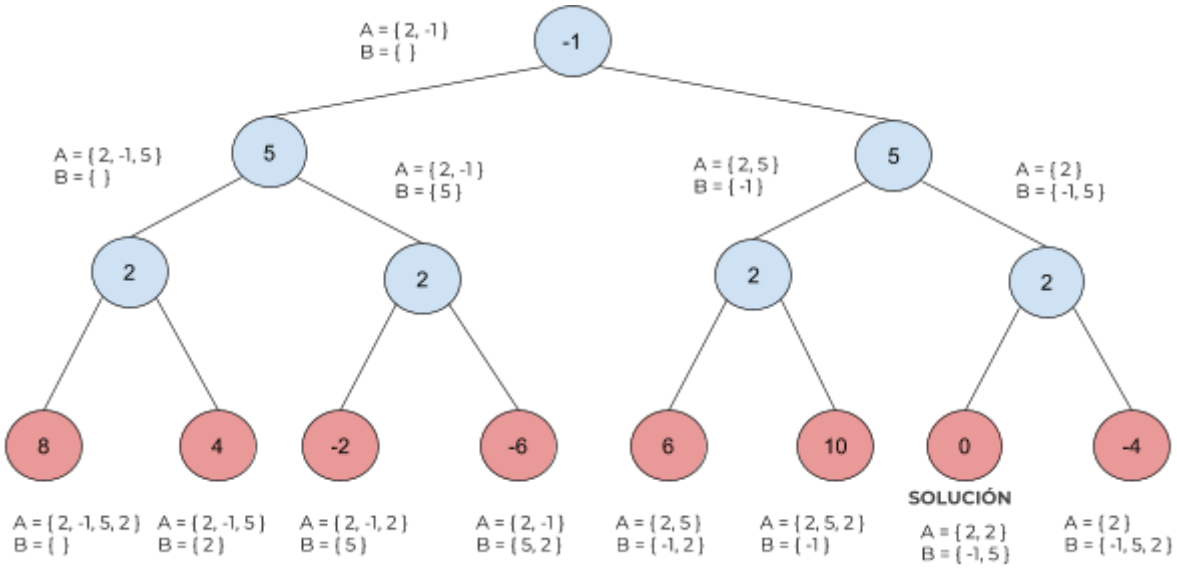
Al volver al inicio del bucle se decrementa decisiones[i] por lo que quedaría decisiones = {0, 1, 1, 0}



i = 3
A = {2, 2
B = {-1, 5 // añadimos al subconjunto A X[i]
i = 3 // no incrementamos i porque es hoja
decisiones = {0, 1, 1, 0}

Al volver al inicio del bucle se decrementa decisiones[i] por lo que quedaría decisiones = {0, 1, 1, -1}. Al ser el valor de decisiones[i] = -1, se produciría backtracking pasaríamos a la rama en la cual añadimos el elemento 5 en el subconjunto A.

Seguiría explorando el árbol hasta hasta obtener el resultado final:



Como vemos encontramos la solución dada en la ejecución del algoritmo en el nodo hoja en el cual los subconjuntos son A = {2, 2} y B = {-1, 5}.