Universidad ORT

Facultad de Ingeniería

Tarea: Reducción entre Problemas NP-Completos
Teoría de la Computación

Ingeniería en Sistemas

Martina Barone - 302285
Ana Clara Vázquez - 282326
Profesores: Damián Ferencz y Mateo Saravia
2025

Parte 1: Verificadores y Reducción en Haskell

1.2 Verificadores en tiempo polinomial

Verificador `verifyA` (para 3-SAT): La función `verifyA` toma una fórmula booleana φ en 3-CNF representada como una lista de m cláusulas, junto con una asignación de valores de verdad para n variables.
La función recorre las cláusulas una por una ($\mathcal{O}(m)$).
Para cada cláusula $c = (t_1, t_2, t_3)$, llama a `clausulaTrue`, que evalúa si al menos uno de los tres literales es verdadero bajo la asignación. Esto implica (en el peor caso) 3 llamadas a `evalLit`.
Cada `evalLit` realiza una búsqueda en la asignación (una lista de largo n) usando `lookup`, lo cual tiene costo $\mathcal{O}(n)$ en el peor caso. Por lo tanto:

$$\text{Costo por cláusula} = 3 \cdot \mathcal{O}(n) = \mathcal{O}(n),$$

$$\text{Costo total} = \mathcal{O}(m \cdot n).$$

$\Rightarrow$ `verifyA` se evalúa en tiempo polinomial respecto al tamaño de la entrada (m, n).

Verificador `verifyB` (para Optimización de Inversiones Internas con Dependencias entre Grupos de Proyectos): La función `verifyB` toma una instancia del problema con p proyectos y g grupos, junto con una solución propuesta que consiste en una lista de s proyectos seleccionados.

La función evalúa tres condiciones:

- `gruposCompletos`: recorre los g grupos. Para cada grupo, verifica si ninguno de sus proyectos está en la solución, o si todos lo están.
  Cada verificación implica recorrer el grupo (supuesto de longitud promedio k) y, para cada proyecto, verificar si pertenece a la solución usando `elem`, que compara contra los s elementos seleccionados.
  Costo total: $\mathcal{O}(g \cdot k \cdot s)$

- `costoTotal`: recorre los p proyectos. Para cada proyecto (n, c, b), verifica si $n \in$ `sol` usando `elem`, que tiene costo $\mathcal{O}(s)$.
  Costo total: $\mathcal{O}(p \cdot s)$

- `beneficioTotal`: mismo análisis que `costoTotal`, ya que realiza la misma verificación de pertenencia.
  Costo total: $\mathcal{O}(p \cdot s)$

Por lo tanto:

$$\text{Costo total} = \mathcal{O}(g \cdot k \cdot s + 2 \cdot p \cdot s)$$

$\Rightarrow$ `verifyB` se evalúa en tiempo polinomial respecto al tamaño de la entrada (p, g, s).

1.4 Reducción polinomial entre problemas

El objetivo de esta sección es definir una estrategia para reducir instancias del problema A (3-SAT) a instancias del problema B (Optimización de Inversiones Internas con Dependencias entre Grupos de Proyectos, o simplemente OP), mediante una reducción polinomial.
Para ello, construiremos una transformación que, dada una instancia de 3-SAT, genere en tiempo polinomial una instancia equivalente del problema OP. Es decir, queremos definir una función:

$$\texttt{reduceAToB} :: DomA \to DomB$$

Dado que 3-SAT y OP son problemas con dominios muy diferentes, realizaremos la reducción en tres etapas:

$$\text{3-SAT} \leq_p \text{Subset Sum} \leq_p \text{Knapsack} \leq_p \text{OP}$$

Cada uno de estos pasos será descrito formalmente y se justificará que cada transformación es computable en tiempo polinomial. En consecuencia, la composición de todas estas transformaciones también es una función computable en tiempo polinomial. Finalmente, se argumentará que dicha composición genera una instancia del problema OP que es satisfacible si y sólo si la instancia original de 3-SAT lo es.

Problemas involucrados

3-SAT Problema de decidir si existe una asignación de valores de verdad a las variables de una fórmula booleana en forma normal conjuntiva (CNF) donde cada cláusula tiene exactamente 3 literales, que haga que la fórmula sea verdadera.

Subset Sum (Suma de Subconjuntos) Dado un conjunto $S = \{x_1, x_2, \ldots, x_n\}$ de enteros positivos y un entero t, determinar si existe un subconjunto $S' \subseteq S$ cuya suma de elementos sea exactamente t.

Knapsack (Mochila) Dado un conjunto de objetos con pesos $w_1, \ldots, w_n$ y valores $v_1, \ldots, v_n$, una capacidad máxima W y un valor objetivo V, decidir si existe un subconjunto de objetos cuya suma de pesos no exceda W y cuya suma de valores sea al menos V.

Problema OP (Optimización de Proyectos con Dependencias Grupales) Dado un conjunto de proyectos, cada uno con un costo, un beneficio y pertenencia a grupos con dependencias, junto con un presupuesto máximo M y un beneficio mínimo V, decidir si existe una selección de proyectos que satisfaga las restricciones de presupuesto, beneficio y dependencias grupales.

Reducción de 3-SAT a Subset Sum

Definición de la reducción Sea la función $\texttt{reducer} : Dom_{\text{3-SAT}} \to Dom_{SubsetSum}$ tal que para toda instancia φ de 3-SAT se cumple:

$$Q_{\text{3-SAT}}(\varphi) \iff Q_{SubsetSum}(\texttt{reducer}(\varphi)) \quad \wedge \quad \text{costo}(\texttt{reducer}) \in O(n^k)$$

Definición de los dominios

$$Dom_{\text{3-SAT}} := \{\varphi \mid \varphi \text{ es una fórmula en 3-CNF}\}$$

$$Dom_{SubsetSum} := ([\mathbb{N}], \mathbb{N}) \quad \text{donde } [\mathbb{N}] \text{ denota una lista finita de números naturales.}$$

Construcción del reductor Sea $\varphi = c_1 \wedge c_2 \wedge \cdots \wedge c_m$, donde cada cláusula $c_i = \ell_{i1} \vee \ell_{i2} \vee \ell_{i3}$ es la disyunción de tres literales ℓ_{ij} (variables o negaciones).
Dada φ, construiremos un conjunto S de números naturales y un entero k tales que:

$$\exists S' \subseteq S \text{ tal que } \sum_{s \in S'} s = k \iff \varphi \text{ es satisfacible.}$$

La construcción se basa en representar cada número en base 10 con $n + m$ dígitos: los primeros n dígitos codifican las variables y los últimos m las cláusulas. Los dígitos se analizan columna por columna sin acarreos.

- Por cada variable x_i se agregan dos números:
  - y_i: representa asignar $x_i = \text{true}$
  - z_i: representa asignar $x_i = \text{false}$

  En la posición i, ambos tienen un 1. Además, en cada cláusula donde x_i (o $\neg x_i$) aparece, el dígito correspondiente a esa cláusula se pone en 1 en y_i o z_i respectivamente.
- Por cada cláusula c_j se agregan dos números auxiliares s_j y t_j, que tienen un 1 únicamente en la posición correspondiente a la cláusula j.
- El número objetivo k tiene:
  - Un 1 en cada posición de variable (para forzar que se elija solo y_i o z_i, pero no ambos).
  - Un 3 en cada posición de cláusula (para forzar que al menos un literal verdadero contribuya a cada cláusula; los números s_j y t_j completan la suma si es necesario).

Ejemplo: sea la fórmula

$$\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$

Con $n = 3$ variables y $m = 2$ cláusulas, cada número tiene 5 dígitos: 3 para variables, 2 para cláusulas.

	x_1	x_2	x_3	c_1	c_2
y_1	1	0	0	1	0
z_1	1	0	0	0	1
y_2	0	1	0	1	1
z_2	0	1	0	0	0
y_3	0	0	1	0	0
z_3	0	0	1	1	1
s_1	0	0	0	1	0
t_1	0	0	0	1	0
s_2	0	0	0	0	1
t_2	0	0	0	0	1
k	1	1	1	3	3

La suma exacta de un subconjunto de estos números será igual a $k = 11133$ si y solo si la fórmula φ es satisfacible. La codificación garantiza:

- Cada variable contribuye con exactamente un 1 en su posición (sólo una elección entre y_i o z_i).
- Las columnas de cláusulas deben sumar exactamente 3: $s_j + t_j = 2$, por lo que al menos un 1 debe provenir de un literal verdadero que aparece en la cláusula.

Justificación de polinomialidad La construcción de S y k se realiza en tiempo polinomial respecto al tamaño de φ.

- Se recorre cada cláusula c_j de φ (hay m en total), y para cada una se procesan sus literales. Esto permite construir los números correspondientes a cada literal (ya sea positivo o negativo), codificando sus ocurrencias en las posiciones de dígito asociadas a variables y cláusulas. Como hay $3m$ literales en total, este paso es de complejidad $\mathcal{O}(m)$.
- Se agregan dos números adicionales por cada cláusula (s_j y t_j), cada uno con un único dígito 1 en la posición correspondiente a la cláusula. Este paso también tiene complejidad $\mathcal{O}(m)$.
- Finalmente, se construye el número objetivo k, que tiene n unos y m treses en su representación en base 10. Este paso tiene complejidad $\mathcal{O}(n + m)$.

En consecuencia, la construcción puede realizarse en tiempo polinomial en el tamaño de la fórmula φ.

Demostración de equivalencia

Demostración de equivalencia Sea `reducer`$(\varphi) = (S, k)$, donde

- φ es una fórmula en 3-CNF con n variables y m cláusulas.
- S es el conjunto de números codificados en base 10, que representan:
  - Los literales verdaderos (por cada variable, y_i o z_i),

  - Los números auxiliares s_j y t_j para completar cláusulas.

- k es el número objetivo, que tiene: Un 1 en cada posición de variable y un 3 en cada posición de cláusula.

1. Si φ es satisfacible, entonces existe una asignación de valores de verdad que satisface todas las cláusulas.

   Por lo tanto, para cada variable x_i, elegimos y_i si $x_i = \text{true}$ o z_i si $x_i = \text{false}$.

   Como cada cláusula es satisfecha, al menos uno de sus literales está presente en la suma.

   Para alcanzar el valor 3 en cada posición de cláusula (en caso de que los literales no alcancen), se agregan los números auxiliares s_j y t_j.

   Así, existe un subconjunto $S' \subseteq S$ tal que $\sum_{s \in S'} s = k$.

2. Si existe un subconjunto $S' \subseteq S$ tal que $\sum_{s \in S'} s = k$, entonces:

   En las primeras n posiciones, la suma debe ser exactamente 1 para cada variable, lo que implica que se eligió exactamente uno entre y_i y z_i por variable, definiendo así una asignación total.

   En las últimas m posiciones, la suma debe ser 3 para cada cláusula, lo que solo es posible si al menos uno de los números elegidos representa un literal verdadero en esa cláusula (los auxiliares s_j y t_j por sí solos no alcanzan).

   Entonces, todas las cláusulas están satisfechas por la asignación definida.

Por lo tanto, se cumple:

$$Q_{\text{3-SAT}}(\varphi) \iff Q_{SubsetSum}(\texttt{reducer}(\varphi)),$$

y concluimos que

$$\text{3-SAT} \leq_p \text{Subset Sum}.$$

Reducción de Subset Sum a Knapsack

Definición de la reducción Sea la función $\texttt{reducer} : Dom_{SubsetSum} \to Dom_{Knapsack}$ tal que, para toda instancia (S, t) de Subset Sum, se cumple:

$$Q_{SubsetSum}(S,t) \iff Q_{Knapsack}(\texttt{reducer}(S,t)) \quad \wedge \quad \text{costo}(\texttt{reducer}) \in \mathcal{O}(n^k)$$

Definición de los dominios

$$Dom_{SubsetSum} := ([\mathbb{N}], \mathbb{N}) \quad \text{donde } [\mathbb{N}] \text{ denota una lista finita de números naturales.}$$

$$Dom_{Knapsack} := ([\text{Objeto}], \mathbb{N}, \mathbb{N}) \quad \text{con Objeto} := (\text{nombre} : \texttt{String}, \text{peso} : \mathbb{N}, \text{valor} : \mathbb{N})$$

Construcción del reductor Dada una instancia (S, t) de Subset Sum con

$$S = \{x_1, x_2, \ldots, x_n\},$$

definimos la instancia de Knapsack como:

$$\texttt{reducer}(S, t) = (\{o_1, o_2, \ldots, o_n\}, W, V)$$

donde para cada $x_i \in S$:

$$o_i = (\text{"item}_i\text{"}, w_i = x_i, v_i = x_i)$$

y se establecen:

$$W = t, \quad V = t$$

Justificación de polinomialidad La construcción recorre la lista $S = \{x_1, x_2, \ldots, x_n\}$ una sola vez, y para cada elemento x_i genera un objeto cuyo peso y valor son iguales a dicho x_i. La capacidad y el beneficio mínimo se asignan directamente a t. Por lo tanto, la reducción se computa en tiempo $\mathcal{O}(n)$.

Demostración de equivalencia Sea $\texttt{reducer}(S, t) = (O, W, V)$ donde

$$O = \{(\text{"item}_i\text{"}, x_i, x_i) \mid x_i \in S, i = 1, \ldots, n\}, \quad W = t, \quad V = t.$$

1. Si existe un subconjunto $S' \subseteq S$ tal que $\sum_{x \in S'} x = t$, entonces existe un subconjunto de objetos $O' \subseteq O$ definido por

   $$O' = \{o_i \in O \mid x_i \in S'\},$$

   cuya suma de pesos y valores cumple

   $$\sum_{o \in O'} \text{peso}(o) = \sum_{o \in O'} \text{valor}(o) = t,$$

   por lo que satisfacen las restricciones de la mochila.

2. Si existe un subconjunto de objetos $O' \subseteq O$ tal que

   $$\sum_{o \in O'} \text{peso}(o) \leq W \quad \text{y} \quad \sum_{o \in O'} \text{valor}(o) \geq V,$$

   entonces, dado que para cada objeto peso y valor coinciden y que $W = V = t$, necesariamente se cumple

   $$\sum_{o \in O'} \text{peso}(o) = \sum_{o \in O'} \text{valor}(o) = t.$$

   Por lo tanto, el conjunto

   $$S' = \{x_i \mid o_i \in O'\}$$

   es un subconjunto de S cuya suma es t.

Así, se cumple:

$$Q_{SubsetSum}(S, t) \iff Q_{Knapsack}(\texttt{reducer}(S, t)).$$

y concluimos que

$$SubsetSum \leq_p Knapsack.$$

Reducción de Knapsack a Optimización con Proyectos (OP)

Definición de la reducción Sea la función $\texttt{reducer} : Dom_{Knapsack} \to Dom_{OP}$ tal que, para toda instancia $(\mathcal{O}, W, V)$ de Knapsack, se cumple:

$$Q_{Knapsack}(\mathcal{O}, W, V) \iff Q_{OP}(\texttt{reducer}(\mathcal{O}, W, V)) \quad \wedge \quad \text{costo}(\texttt{reducer}) \in \mathcal{O}(n^k)$$

Definición de los dominios

$$Dom_{Knapsack} := ([\text{Objeto}], \mathbb{N}, \mathbb{N}) \quad \text{con Objeto} := (\text{nombre} : \texttt{String}, \text{peso} : \mathbb{N}, \text{valor} : \mathbb{N})$$

$$Dom_{OP} := ([\text{Proyecto}], [\text{Grupo}], \mathbb{N}, \mathbb{N})$$

$$\begin{aligned} \text{donde} \quad \text{Proyecto} &:= (\text{nombre} : \texttt{String}, \text{costo} : \mathbb{N}, \text{beneficio} : \mathbb{N}) \\ \text{Grupo} &\subseteq \mathcal{P}(\{\text{nombres de proyectos}\}) \\ M &\in \mathbb{N} \quad \text{(presupuesto máximo)} \\ B &\in \mathbb{N} \quad \text{(beneficio mínimo)} \end{aligned}$$

Construcción del reductor Dada una instancia $(\mathcal{O}, W, V)$ de Knapsack, con

$$\mathcal{O} = \{(\text{"item}_i\text{"}, w_i, v_i) \mid i = 1, \ldots, n\},$$

definimos la instancia de Optimización de Proyectos como:

$$\texttt{reducer}(\mathcal{O}, W, V) = (\mathcal{P}, \mathcal{G}, M, B),$$

donde

$$\mathcal{P} = \{(\text{"item}_i\text{"}, w_i, v_i) \mid i = 1, \ldots, n\},$$
$$\mathcal{G} = \{\{\text{"item}_i\}\} \mid i = 1, \ldots, n\},$$

y

$$M = W \quad , \quad B = V$$

Cada objeto se mapea a un proyecto con costo igual al peso y beneficio igual al valor del objeto, y se asigna un grupo trivial que contiene únicamente ese proyecto. De esta forma, no se generan dependencias entre proyectos y la restricción de grupos se cumple automáticamente para cualquier selección de proyectos.

Justificación de polinomialidad La construcción recorre la lista de objetos una única vez, generando un proyecto y un grupo trivial para cada objeto. Las asignaciones del presupuesto y beneficio se realizan de forma directa. Por lo tanto, la función `reducer` se computa en tiempo $\mathcal{O}(n)$.

Demostración de equivalencia Sea $\texttt{reducer}(\mathcal{O}, W, V) = (\mathcal{P}, \mathcal{G}, M, B)$ con $M = W$ y $B = V$. Se verifica que:

1. Si existe un subconjunto $\mathcal{O}' \subseteq \mathcal{O}$ tal que

$$\sum_{o \in \mathcal{O}'} \text{peso}(o) \leq M \quad \text{y} \quad \sum_{o \in \mathcal{O}'} \text{valor}(o) \geq B,$$

entonces

$$\mathcal{P}' = \{(\text{nombre}(o), \text{peso}(o), \text{valor}(o)) \mid o \in \mathcal{O}'\} \subseteq \mathcal{P}$$

es una selección válida de proyectos en OP que satisface:

- Cada proyecto pertenece a un único grupo, dado que los grupos son triviales y disjuntos,
- La restricción de costo total $\leq M$,
- El beneficio mínimo $\geq B$.

2. Recíprocamente, si existe $\mathcal{P}' \subseteq \mathcal{P}$ que cumple las restricciones de OP, entonces

$$\mathcal{O}' = \{o \in \mathcal{O} \mid \text{nombre}(o) \in \{\text{nombre}(p) \mid p \in \mathcal{P}'\}\}$$

es solución válida para Knapsack con peso total $\leq W$ y beneficio $\geq V$.

Por lo tanto,

$$Q_{Knapsack}(\mathcal{O}, W, V) \iff Q_{OP}(\texttt{reducer}(\mathcal{O}, W, V)),$$

y concluimos que

$$Knapsack \leq_p OP.$$

Composición de las reducciones

Para formalizar la reducción de 3-SAT al problema OP, definimos las reducciones correspondientes a cada paso intermedio:

$$\texttt{reducer}_1 : Dom_{\text{3-SAT}} \to Dom_{SubsetSum},$$
$$\texttt{reducer}_2 : Dom_{SubsetSum} \to Dom_{Knapsack},$$
$$\texttt{reducer}_3 : Dom_{Knapsack} \to Dom_{OP}.$$

La reducción compuesta

$$\texttt{reduceAToB} := \texttt{reducer}_3 \circ \texttt{reducer}_2 \circ \texttt{reducer}_1 : Dom_{\text{3-SAT}} \to Dom_{OP}$$

es una reducción polinomial válida, ya que la composición de reducciones polinomiales también es polinomial.
En consecuencia, dado que

$$\text{3-SAT} \leq_p \text{Subset Sum} \leq_p \text{Knapsack} \leq_p OP,$$

existe una función `reduceAToB` que transforma cualquier instancia φ de 3-SAT en una instancia equivalente de OP, cumpliendo

$$Q_{\text{3-SAT}}(\varphi) \iff Q_{OP}(\texttt{reduceAToB}(\varphi)).$$

Por lo tanto, concluimos que

$$\text{3-SAT} \leq_p OP.$$

Parte 2: Verificadores y Reducción en otros modelos

2.1 Verificador de 3-SAT en Imp

Para abordar la implementación solicitada, partimos de la función `verifyA` definida en Haskell. Dado que esta función es recursiva, se vio necesario recurrir a una eliminación de la recursión para poder implementarla en `Imp`, un lenguaje imperativo.
El modelo imperativo en su versión primitiva o pura no posee mecanismos de recursión, los cuales son característicos del modelo funcional. Por esta razón, cuando se traslada un algoritmo desde un entorno funcional hacia uno imperativo, resulta común reemplazar la recursión por iteración explícita mediante estructuras como bucles y acumuladores.

El primer paso fue implementar una versión *tail-recursiva* de `verifyA` en Haskell. Una función es *tail-recursiva* cuando su llamada recursiva es la última operación que realiza, lo que facilita su traducción a una estructura iterativa.

A continuación se muestra dicha versión de `verifyA`. En ella se utiliza un acumulador booleano que almacena el resultado parcial de la verificación, mientras se recorre la lista de cláusulas pendientes por procesar. La evaluación de cada cláusula se realiza accediendo a la valuación, que se mantiene disponible y sin cambios a lo largo de todo el algoritmo.

```
verifyA :: (DomA, SolA) -> Bool
verifyA(clausulas, valuacion) = verifyATR True (clausulas, valuacion)
    where verifyATR :: Bool -> (DomA, SolA) -> Bool
        verifyATR b ([], _) = b
        verifyATR b ((t1, t2, t3):ps, valuacion) =
            let clausulaTrue = evalLit t1 valuacion ||
                               evalLit t2 valuacion ||
                               evalLit t3 valuacion
            in verifyATR (clausulaTrue && b) (ps, valuacion)

evalLit :: Termino -> SolA -> Bool
evalLit (var, val) asignacion = case lookup var asignacion of
    Just b -> b == val
    Nothing -> False
```

Ahora se presenta la traducción al lenguaje Imp de esta versión *tail-recursiva*, adaptando la recursión a una estructura iterativa.
Para la implementación se realizaron dos consideraciones importantes:

- Se asume que las cláusulas se representan mediante el constructor `Clausula[t1, t2, t3]`, y cada término como `Termino[var, val]`.
- Se asume que la valuación contiene todas las variables que aparecen en las cláusulas, no se consideran fallos en la operación `lookup`.

```
VERIFY_SAT(clausulas, valuacion){
  -- se asume la definicion de lookup
  def igualdad(b1, b2) returns r {
    case b1 of [
      True -> [], {
```

```
      case b2 of [
        True -> [], { r := True },
        False -> [], { r := False }
      ]
    },
    False -> [], {
      case b2 of [
        True -> [], { r := False },
        False -> [], { r := True }
      ]
    }
  ]
}
def evalClausula(clausula, valuacion) returns r {
  case clausula of [
    Clausula -> [t1, t2, t3], {
      case t1 of [
        Termino -> [var1, val1], {
          lookup(valuacion, var1) returns on valor1;
          igualdad(valor1, val1) returns on r1;
          case r1 of [
            True -> [], { r := True },
            False -> [], {
              case t2 of [
                Termino -> [var2, val2], {
                  lookup(valuacion, var2) returns on valor2;
                  igualdad(valor2, val2) returns on r2;
                  case r2 of [
                    True -> [], { r := True },
                    False -> [], {
                      case t3 of [
                        Termino -> [var3, val3], {
                          lookup(valuacion, var3) returns on valor3;
                          igualdad(valor3, val3) returns on r3;
                          case r3 of [
                            True -> [], { r := True },
                            False -> [], { r := False }
                          ]
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```

```
  def and(b1, b2) returns r {
    case b1 of [
      True -> [], {
        case b2 of [
          True -> [], { r := True },
          False -> [], { r := False }
        ]
      },
      False -> [], { r := False }
    ]
  }
  local acc, ps, clausulaTrue {
    acc, ps := True[], clausulas
    while ps is [
      : -> [p, ps'], {
        evalClausula(p, valuacion) returns on clausulaTrue;
        and(clausulaTrue, acc) returns on acc;
        ps := ps';
      }
    ];
    res := acc
  }
}
```

2.2 Codificación de la Reducción en Máquinas de Turing

En esta sección se define un alfabeto Σ y se presenta una codificación de las instancias de *DomA* y *DomB* como cadenas sobre Σ. Posteriormente, se especifica de manera esquemática una Máquina de Turing que, dada como entrada la codificación de una instancia de *DomA*, produce como salida la codificación de una instancia correspondiente de *DomB*, preservando la relación de satisfacibilidad entre ambas.

Codificación de instancias de *DomA*

Sea el alfabeto finito

$$\Sigma = \{\, \#, \texttt{T}, \texttt{F}, \texttt{AND}, \texttt{OR}, x_1, x_2, \ldots, x_n \,\}$$

que incluye:

- El símbolo # como delimitador de principio y fin.
- Los identificadores `T` y `F`, que representan los valores booleanos True y False respectivamente.
- Los separadores `OR` y `AND`, que se utilizan para separar términos dentro de una cláusula y cláusulas entre sí respectivamente.
- Un conjunto finito de identificadores de variables proposicionales: $x_1, x_2, \ldots, x_n$.

Dada una fórmula en 3-CNF con k cláusulas, cada una de la forma

$$\big((v_{i,1}, b_{i,1}), (v_{i,2}, b_{i,2}), (v_{i,3}, b_{i,3})\big),$$

donde $i = 1, \ldots, k$, indica el numero de cláusula, $v_{i,j} \in \{x_1, x_2, \ldots, x_n\}$ representa una variable y $b_{i,j} \in \{\texttt{T}, \texttt{F}\}$ indica si el literal es positivo o negado.

La codificación de dicha fórmula como cadena sobre Σ es la siguiente:

$$|\cdots|\#|v_{1,1}|b_{1,1}|\texttt{OR}|v_{1,2}|b_{1,2}|\texttt{OR}|v_{1,3}|b_{1,3}|\texttt{AND}|\cdots|\texttt{AND}|v_{k,1}|b_{k,1}|\texttt{OR}|v_{k,2}|b_{k,2}|\texttt{OR}|v_{k,3}|b_{k,3}|\#|\cdots|$$

Codificación de instancias de $DomB$

Sea el alfabeto finito

$$\Sigma = \{\ \#, \texttt{P}, \texttt{G}, \texttt{CMAX}, \texttt{BMIN}, p_1, p_2, \ldots, p_n\ , 0, 1, \ldots, 9\}$$

que incluye:

- El símbolo # como delimitador entre elementos y como demilitador de principio y fin.
- El identificador `P`, que marca el comienzo de la sección (listado) de proyectos.
- El identificador `G`, que marca el comienzo de la sección (listado) de grupos.
- Los identificadores `CMax` y `BMin`, que preceden el presupuesto máximo permitido y el beneficio mínimo requerido respectivamente.
- Los símbolos $p_1, p_2, \ldots, p_n$ que representan nombres de proyectos (asumidos distintos de las palabras clave `P`, `G`, etc.).
- Los dígitos del 0 al 9 que se utilizan para representar números naturales (costos, beneficios, límites).

Dada una instancia compuesta por:

- Una lista de proyectos (p_i, c_i, b_i), donde p_i es el nombre del proyecto, c_i es su costo y b_i es su beneficio.
- Una lista de grupos, representados como listas de nombres de proyectos $p_1, p_2, \ldots, p_n$.
- Un presupuesto máximo C y un beneficio mínimo requerido B, ambos números naturales

La codificación como cadena sobre Σ es la siguiente:

$$\begin{aligned}&|\cdots|\,\#\,|P|\,p_1\,|\,c_1\,|\,b_1|\,\#\,|p_2|\,c_2\,|\,b_2|\,\#\,|\ldots\ |p_n|\,c_n\,|\,b_n|\,\#\,|\\ &G|\,g_1|\,\#\,|g_2|\,\#\,|\ldots\ |g_m|\#\,|\text{CMax}|\,C\,|\,\text{BMin}|\,B\,|\,\#\,|\cdots|\end{aligned}$$

donde cada g_i representa una secuencia de nombres de proyectos que forman un grupo dependiente (es decir, un p_i por espacio en cinta).

Especificación de la Máquina de Turing

Sea el alfabeto:

$$\Sigma = \{\ \#, \texttt{T}, \texttt{F}, \texttt{AND}, \texttt{OR}, x_1, x_2, \ldots, x_n, \texttt{P}, \texttt{G}, \texttt{CMAX}, \texttt{BMIN}, p_1, p_2, \ldots, p_n, 0, 1, \ldots, 9\}$$

Entrada (init):

$$|\cdots|\,\#\,|v_{1,1}|\,b_{1,1}|\,\texttt{OR}\,|v_{1,2}|\,b_{1,2}|\,\texttt{OR}\,|v_{1,3}|\,b_{1,3}|\,\texttt{AND}\,|\cdots|\,\texttt{AND}\,|v_{k,1}|\,b_{k,1}|\,\texttt{OR}\,|v_{k,2}|\,b_{k,2}|\,\texttt{OR}\,|v_{k,3}|\,b_{k,3}|\overset{V}{\#}\,|\cdots|$$

Salida (halt):

$$|\cdots|\,\#\,|v_{1,1}|\,b_{1,1}|\,\texttt{OR}\,|v_{1,2}|\,b_{1,2}|\,\texttt{OR}\,|v_{1,3}|\,b_{1,3}|\,\texttt{AND}\,|\cdots|\,\texttt{AND}\,|v_{k,1}|\,b_{k,1}|\,\texttt{OR}\,|v_{k,2}|\,b_{k,2}|\,\texttt{OR}\,|v_{k,3}|\,b_{k,3}|\,\#\,|$$

$$|P|\,p_1\,|\,c_1\,|\,b_1\,|\,\#\,|p_2|\,c_2\,|\,b_2\,|\,\#\ \ldots\ |p_n|\,c_n\,|\,b_n\,|\,\#\,|G|\,g_1\,|\,\#\,|g_2|\,\#\ \ldots\ |g_m|\,\#\,|\text{CMax}|\,C\,|\,\text{BMin}|\,B\,|\overset{V}{\#}\,|\cdots|$$

El funcionamiento de la máquina se basa en la simulación de la función `reduceAToB`, cuya estrategia fue definida en la parte 1.4.

Fuentes consultadas

- Explicación sobre cómo reducir 3-SAT a Subset Sum.
  https://cs.stackexchange.com/questions/107376/reduction-from-3-sat-to-subset-sum-problem
- Ejemplo de reducción de 3-SAT a Subset Sum.
  https://www.youtube.com/watch?v=i8Kt9IBZ8FU
- Discusión sobre cómo reducir Subset Sum a Knapsack.
  https://cs.stackexchange.com/questions/133211/how-to-prove-that-the-subset-sum-problem-is-polynomially-reducible-to-the-knapsa
- Explicación general sobre reducciones entre 3-SAT, Subset Sum y Knapsack en el contexto de NP-completitud
  https://gtalgorithms.wordpress.com/np-knapsack/