

# Trabajo Final

Teoria de la Computacion

Federico Vareika, Guillermo Golpe

2 de Julio 2025

# Ejercicio 1

## 1.4 Reducción reduceAtoB

Para obtener **reduceAtoB**, debemos encontrar un plan de reducciones que reduce **3-SAT** al problema **B**. En nuestra investigación logramos obtener el siguiente plan de reducciones:

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

### Reducción $\mathbf{3-SAT} \leq_p \mathbf{Subset Sum}$

Cabe aclarar que esta reducción no fue ideada por nosotros, sino que tomamos como referencia la reducción mostrada en [1].

Para realizar esta reducción tenemos que moldar el problema **3-SAT** para que se pueda reducir de manera trivial a **Subset Sum**.

Tomamos el problema **3-SAT** como un conjunto de variables  $x_1, \dots, x_n$  y un conjunto de clausulas  $c_1, \dots, c_r$ . Para cada variable  $x_i$ , se puede decir que tenemos dos opciones, que esta se evalúe a **True** o que se evalúe a **False**, llamaremos a estas opciones como  $a_i$  y  $b_i$  respectivamente. Si se toma la opción  $b_i$ , entonces podemos concluir que cada clausula que tenga a la variable  $x_i$  negada se evaluará a **True**.

Si ahora tomamos estas opciones como numeros decimales de manera que:

$$a_i = b_i = 10^{n-i}$$

Entonces podemos formar la siguiente tabla tomando  $n = 3$ :

	$x_1$	$x_2$	$x_3$
$a_1$	1	0	0
$b_1$	1	0	0
$a_2$	0	1	0
$b_2$	0	1	0
$a_3$	0	0	1
$b_3$	0	0	1

En **3-SAT** solo podemos tomar una opcion por variable, y debemos definir todas las variables. Dado esto, si sumamos todas las opciones elegidas, y llamamos esta suma  $k$ :

	$x_1$	$x_2$	$x_3$
$a_1$	1	0	0
$b_1$	1	0	0
$a_2$	0	1	0
$b_2$	0	1	0
$a_3$	0	0	1
$b_3$	0	0	1
$k$	1	1	1

Se eligen las opciones en verde como ejemplo.

Teniendo esto y usando el hecho que si elegimos  $a_i$  entonces cada clausula que contenga  $x_i$  no negada se va a evaluar a **True**, podemos extender esta tabla. Para el ejemplo anterior tomamos la siguiente expresion:

$$\underbrace{(x_1 \vee \neg x_2 \vee x_3)}_{c_1} \wedge \underbrace{(\neg x_1 \vee \neg x_2 \vee \neg x_3)}_{c_2}$$

Y derivamos la tabla:

	$x_1$	$x_2$	$x_3$	$c_1$	$c_2$
$a_1$	1	0	0	1	0
$b_1$	1	0	0	0	1
$a_2$	0	1	0	0	0
$b_2$	0	1	0	1	1
$a_3$	0	0	1	1	0
$b_3$	0	0	1	0	1
$k$	1	1	1	0	3

Se puede ver en esta tabla como indicamos con un 1 las clausulas que se evaluan a **True** al elegir la opcion en esa fila, e indicamos con 0 el resto. Tambien extendimos  $k$  para incluir la suma de las opciones marcadas.

Podemos saber que se cumple **3-SAT** con las opciones elegidas sii se elige solo una opcion de cada variable 1 y todas las clausulas son evaluadas a **True** al menos una vez 2.

Estas reglas se ven cumplidas en los digitos de  $k$ :

$$digitoK(i) = \begin{cases} 1 & \text{si } i \in [1, n] \\ v & \text{si } i \in [n+1, n+r], \quad v \geq 1 \end{cases} \quad (1)$$

Para el ejemplo anterior, es claro que una solucion es  $x_1 = \mathbf{True}$ ,  $x_2 = \mathbf{False}$ ,  $x_3 = \mathbf{True}$ . Por lo tanto, deberiamos poder comprobarlo con estas nuevas reglas:

	$x_1$	$x_2$	$x_3$	$c_1$	$c_2$
$a_1$	1	0	0	1	0
$b_1$	1	0	0	0	1
$a_2$	0	1	0	0	0
$b_2$	0	1	0	1	1
$a_3$	0	0	1	1	0
$b_3$	0	0	1	0	1
$k$	1	1	1	3	1

$$k = 11131 \Rightarrow k[i] = digitoK(i) \quad \forall i \in [1, n+r]$$

Podemos ver como nos estamos acercando a **Subset Sum**:

- Tenemos una entrada de numeros ( $a_i$  y  $b_i$ ).
- Sumamos un subconjunto de estos numeros.
- Verificamos que esta suma cumple ciertas reglas.

El ultimo paso tendria que ser "verificamos que esta suma es igual a un numero  $k'$ ". Para esto, debemos modificar las reglas originales:

$$digitoK'(i) = \begin{cases} 1 & \text{si } i \in [1, n] \\ 3 & \text{si } i \in [n+1, n+r] \end{cases} \quad (3)$$

$$(4)$$

Y agregamos nuevos elementos al conjunto de numeros. Estos van a ser dos por clausula, de tal manera que se podra 'rellenar' de cierta forma el digito de esta clausula. Llamaremos a estos numeros  $s_i$  y  $t_i$ . Podemos ver estos nuevos numeros en la siguiente tabla:

	$x_1$	$x_2$	$x_3$	$c_1$	$c_2$
$a_1$	1	0	0	1	0
$b_1$	1	0	0	0	1
$a_2$	0	1	0	0	0
$b_2$	0	1	0	1	1
$a_3$	0	0	1	1	0
$b_3$	0	0	1	0	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

Ahora podemos elegir los nuevos numeros para rellenar el ultimo digito de  $k$ :

	$x_1$	$x_2$	$x_3$	$c_1$	$c_2$
$a_1$	1	0	0	1	0
$b_1$	1	0	0	0	1
$a_2$	0	1	0	0	0
$b_2$	0	1	0	1	1
$a_3$	0	0	1	1	0
$b_3$	0	0	1	0	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

Y ahora:

$$k = 11133 \Rightarrow k[i] = \text{digito}K'(i) \quad \forall i \in [1, n+r]$$

PPor lo tanto, ya podemos cambiar la tercer regla de 1.4 por:

- Verificamos que la suma es igual a  $\underbrace{1\dots1}_n \underbrace{3\dots3}_r$

Con esto, la reduccion esta completa. Se puede ver como crear el conjunto de numeros  $a_i$  y  $b_i$  es recorrer la lista de clausulas por cada variable ( $O(n*r)$ ), y crear el conjunto de numeros  $s_i$  y  $t_i$  es recorrer la lista de clausulas ( $O(r)$ ), por lo tanto es una reduccion polinomial.

Formalmente, el dominio de **3-SAT** ([Variables], [Clausulas]) se reduce a el dominio de **Subset Sum** ([Numeros],  $k$ )

## Reducción $\text{Subset Sum} \leq_p \text{0-1 Knapsack}$

Esta reducción es mas trivial.

Tenemos el dominio de **Subset Sum** ( $[\text{Numeros}], k$ ) y queremos reducirlo al dominio de **0-1 Knapsack** ( $[\text{Valor}], [\text{Peso}], \text{Valor}, \text{Peso}$ ), donde la solución es un subconjunto de índices  $I$  donde si tenemos la entrada (`valores pesos valorMinimo pesoMaximo`)

$$\sum_{i \in I} \text{valores}[i] \geq \text{valorMinimo}$$
$$\sum_{i \in I} \text{pesos}[i] \leq \text{pesoMaximo}$$

Para reducir **Subset Sum** a **0-1 Knapsack**, debemos unicamente pasar repetir el dominio:

$$\text{solveSubsetSum } \text{nums } k \Rightarrow \text{solve01Knapsack } \text{nums } \text{nums } k \ k$$

Esto va a dar que

$$\sum_{i \in I} \text{nums}[i] \geq k$$
$$\sum_{i \in I} \text{nums}[i] \leq k$$

Por lo tanto:

$$\sum_{i \in I} \text{nums}[i] = k$$

Esto tiene orden de ejecución  $O(1)$ , por lo tanto es una reducción polinomial.

### Reducción 0-1 Knapsack $\leq_p$ B

Esta reduccion es de nuevo sumamente trivial.

Dado el dominio de **0-1 Knapsack** ([Valor], [Peso], Valor, Peso) y el dominio de **B** ([Proyecto], [Grupo], c, b, Beneficio, Costo), el unico problema a resolver que hay es que se hace con los grupos. Esto se da ya que se puede obtener el costo y beneficio de cada proyecto con las funciones  $c : Proyecto \rightarrow Costo$  y  $b : Proyecto \rightarrow Beneficio$ .

Se va a generar un grupo por proyecto, tal que:

$$g_i = p_i \quad \forall i \in [1, n]$$

Dado esto, podemos reducir el resto de los elementos:

$$p_i = "p_i"$$

$$c(p_i) = pesos[i]$$

$$b(p_i) = valores[i] \quad \forall i \in [1, n]$$

El valor y el peso se reducen a el beneficio y el costo respectivamente.

Con estas reducciones demostramos que

$$\mathbf{3-SAT} \leq_p \text{Subset Sum} \wedge \text{Subset Sum} \leq_p \text{0-1 Knapsack} \Rightarrow \mathbf{3-SAT} \leq_p \text{0-1 Knapsack}$$

$$\mathbf{3-SAT} \leq_p \text{0-1 Knapsack} \wedge \text{0-1 Knapsack} \leq_p \mathbf{B} \Rightarrow \mathbf{3-SAT} \leq_p \mathbf{B}$$



## Ejercicio 2

### 2.5 Verificador de 3-SAT en Imp

Para programar en Imp el verificador de SAT, es buena idea primero programarlo en Haskell con el uso de *tail recursion*. Para esto, usamos los tipos definidos para el ejercicio 1.

```
verifyTerm :: Term -> SolA -> Bool
verifyTerm (var, modifier) sol =
    let maybeVal = lookup var sol in
    case maybeVal of
        Just val -> val == modifier

verify3SatTR :: DomA -> SolA -> Bool -> Bool
verify3SatTR [] sol acc = acc
verify3SatTR (c@(t1, t2, t3):cs) sol acc =
    let terms = [t1, t2, t3] in
    let verifiedClause = foldr (\t -> (verifyTerm t sol ||)) False terms in
    verify3SatTR cs sol (acc && verifiedClause)
```

### Traducido a Imp:

```
VERIFY_SAT(clausulas, validacion) = {  
  local acc {  
    acc := True [];  
    while clausulas is [  
      : -> [c, cs], {  
        case c of [  
          Phi -> [t1, t2, t3], {  
            local clauseVeridity {  
              clauseVeridity := False [];  
              VERIFY_TERM(t1, validacion);  
              OR(res, clauseVeridity);  
              clauseVeridity := res;  
              VERIFY_TERM(t2, validacion);  
              OR(res, clauseVeridity);  
              clauseVeridity := res;  
              VERIFY_TERM(t3, validacion);  
              OR(res, clauseVeridity);  
              clauseVeridity := res;  
  
              AND(acc, clauseVeridity);  
              acc := res;  
            }  
          }  
        ];  
        clausulas := cs  
      }  
    ];  
    res := acc;  
  }  
}
```

Los macros helper:

```
VERIFY_TERM(t, validacion) = {
  local result {
    case t of [
      Term -> [x, b], {
        lookup(validacion, x) on result;
        case b of {
          False -> [], { NOT(res); result := res; }
        }
      }
    ];
    res := result;
  }
}

OR(a, b) = {
  case a of [
    True -> [], {res := a},
    False -> [], {res := b},
  ]
}

AND(a, b) = {
  case a of [
    True -> [], {res := b},
    False -> [], {res := a},
  ]
}

NOT(a) = {
  case a of [
    True -> [], {res := False []},
    False -> [], {res := True []},
  ]
}
```

## References

- [1] Park City Mathematics Institute. *Lecture Notes B07*. 2007. URL: <https://people.clarkson.edu/~alexis/PCMI/Notes/lectureB07.pdf>.