Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de la Universidad

Martín E. Bravo, Franco González y Felipe Avendaño A.

Tema a tratar

Integrantes:	Integrante 1
	Integrante 2
Profesor:	Profesor 1
Auxiliar:	Auxiliar 1
Ayudantes:	Ayudante 1
	Ayudante 2

Fecha: 17 de abril de 2023
Santiago de Chile

Pregunta 1

1.

Se tiene un algoritmo A de caja negra de resolución SAT, es decir, un dispositivo que toma una fórmula de lógica proposicional ϕ, A(ϕ) es verdadero si ϕ es satisfacible.

a) Algoritmo que utiliza A como subrutina para determinar si ϕ es una tautología:

```
def esTautologia(p):
    if A(~p) == 0: #Si ~p es insatisfacible
        return True
    else:
        return False
```

Probaremos si el algoritmo anterior es correcto:

Demostración. PDQ: $\neg\phi$ es insatisfactible $\Leftrightarrow \phi$ es tautología

$$\begin{aligned}\neg\phi \text{ es insatisfactible} &\Leftrightarrow (\emptyset \cup \{\neg\phi\}) \text{ es insatisfactible}\\ &\Leftrightarrow \emptyset \models \phi\\ &\Leftrightarrow \sigma(\phi) = 1\\ &\Leftrightarrow \phi \text{ es Tautología}\end{aligned}$$

□

b) Se tienen 2 fórmulas proposicionales ϕ y φ, se va a determinar si tienen los mismos valores de verdad. Algoritmo que utiliza A para responder a esta pregunta:

```
def equivalentes(p, q):
    if A(~((~p or q) and (p or ~q))) == 0:
        return True
    else:
        return False
```

Probaremos si el algoritmo anterior es correcto:

Demostración. PDQ: $\neg((\neg\phi \vee \varphi) \wedge (\phi \vee \neg\varphi))$ es insatisfactible $\Leftrightarrow \phi \equiv \varphi$

$$\begin{aligned}\neg((\neg\phi \vee \varphi) \wedge (\phi \vee \neg\varphi)) \text{ es insatisfactible} &\Leftrightarrow (\neg\phi \vee \varphi) \wedge (\phi \vee \neg\varphi) \text{ es Tautología}\\ &\Leftrightarrow (\phi \Rightarrow \varphi) \wedge (\varphi \Rightarrow \phi) \text{ es Tautología}\\ &\Leftrightarrow (\phi \Leftrightarrow \varphi) \text{ es Tautología}\\ &\Leftrightarrow \sigma(\phi) = \sigma(\varphi)\\ &\Leftrightarrow \phi \equiv \varphi\end{aligned}$$

□

c) Se tiene una fórmula proposicional ϕ con n variables que se sabe que es satisfacible. Algoritmo que utiliza A como subrutina para obtener una asignación satisfactoria para ϕ utilizando como máximo n llamadas a A:

Sean $\phi_1, ..., \phi_n$ proposiciones de la fórmula, luego sea ϕ' tal que $\sigma(\phi_1) = 1$

Si $A(\phi')$ satisfacible $\Rightarrow \sigma(\phi_1) = 1$
Si $A(\phi')$ insatisfacible $\Rightarrow \sigma(\phi_1) = 0$

Luego ϕ'' tal que $\sigma(\phi_1) = \{\text{valor con el que es satisfacible}\}$ y $\sigma(\phi_2) = 1$, se repite el proceso anterior n veces hasta encontrar todas las valuaciones $\sigma(\phi_k)$, $k \in 1, ..., n$.

Demostración. Para probar la correctitud se procederá por el principio de inducción:

$\boxed{\text{Caso Base:}}$

Sea ϕ una formula lógica con ϕ_1 su única proposición y se sabe que existe $\sigma(\phi_1)$ tal que $\sigma(\phi) = 1$. Sea entonces ϕ' tal que $\sigma(\phi_1) = 1$, como la fórmula depende solo de ϕ_1, si con $\sigma(\phi) = 1$ se cumple que $A(\phi')$ entrega satisfacible también se cumple que $\sigma(\phi') = 1$, por el contrario si $A(\phi')$ es insatisfacible entonces basta con tomar $\sigma(\phi) = 0$, con esa valuación se tendría el otro caso.

$\boxed{\text{Caso Inductivo:}}$

Supongamos que existen n-1 valuaciones $\sigma(\phi_1), ..., \sigma(\phi_{n-1})$ con las que se cumple $A(\phi^{(n-1)})$ entrega satisfacible. Como se sabe que $\phi^{(n-1)}$ es satisfacible entonces debe existir una valuación $\sigma(\phi_n)$ tal que $\sigma(\phi^{(n-1)}) = 1$. Entonces para encontrar la n-ésima valuación basta con tomar $\phi^{(n)}$ tal que $\sigma(\phi_n) = 1$, si con esa valuación $A(\phi^{(n)})$ entrega que es satisfacible entonces encontramos todas las valuaciones con las que $\sigma(\phi) = 1$. En caso de que $A(\phi_{(n)})$ entregue insatisfacible, basta tomar $\sigma(\phi_n) = 0$.

$\square$

El algoritmo anterior nos entrega las n valuaciones de las variables de la formula proposicional con solo n llamadas.

2.

Notemos que $\sigma(a_i) = \sigma(b_i)$ es un factor importante al momento de sumar los números binarios, por lo que necesitamos definir una función auxiliar que indique si esto se cumple o no. Esto se consigue directamente de reescribir y manipular a_i XOR b_i:

$$iguales_i = \neg(\neg a_i \vee \neg b_i) \vee \neg(a_i \vee b_i)$$

Donde $\sigma(iguales_i) = 1$ si y solo si $\sigma(a_i) = \sigma(b_i)$.

Otro factor importante es cuando ocurre lo que denominamos 'arrastre', lo cual ocurre solo de las siguentes formas para este caso: $(1 + 1 = 0$ con arrastre) y $(1 + 1 + 1 = 1$ con arrastre). Lo que hace el arrastre, en esencia, es que si ocurren una de las sumas anteriormente descritas entonces a la siguiente posición se la suma un 1 extra. Definimos lógicamente si es que se 'arrastró' o no un 1 desde la posición i-1 hasta la posición i como la función auxiliar siguente:

$$arrastre_i = \neg(\phi_{i-1} \vee iguales_{i-1}) \vee \neg(\neg a_{i-1} \vee \neg b_{i-1})$$

Donde $\sigma(arrastre_i) = 1$ si y solo si se arrastró un 1 desde la posición i-1 hacia la posición i.

Dadas las definiciones anteriores, construimos nuestras fórmulas ϕ_i, distinguendo los siguentes casos:

$\boxed{\text{Caso inicial}}$ (i=0) Para este caso, $\sigma(\phi_i) = 1$ ssi $\sigma(a_i) \neq \sigma(b_i)$

$$\Rightarrow \phi_i = \neg iguales_i \ , \ donde \ i = 0$$

$\boxed{\text{Caso intermedio}}$ $(i \in \{1, ..., n-1\})$ (Dado $n \geq 2$) Necesitamos que $\sigma(\phi_i) = 1$ ssi $\sigma(a_i) \neq \sigma(b_i)$ y que no haya ocurrido algun arrastre hacia la posición i, o que sí haya ocurrido arrastre hacia la posición i, pero que se cumpla que $\sigma(a_i) = \sigma(b_i)$. Es decir:

$$\phi_i = \neg(iguales_i \vee arrastre_i) \vee \neg(\neg arrastre_i \vee \neg iguales_i) \ , \ donde \ i \in \{1, ..., n-1\}$$

$\boxed{\text{Caso final}}$ (i=n): No existen a_n ni b_n para compararlos entre ellos, y notamos que $\phi_{i=n}$ depende unicamente del arrastre que haya ocurrido desde la posición anterior, de la siguiente forma:

$$\phi_i = arrastre_i \ , \ donde \ i = n$$

Con esto se han construido, para todo $i \in \{0, 1, ..., n-1, n\}$, los ϕ_i tales que: $\sigma(\phi_i) = 1$ ssi el i-ésimo bit de la suma es un 1. Notemos que las construcciones fueron expresadas solo a base de los operadores $\{\neg, \vee\}$ (incluyendo las funciones auxiliares, las cuales también fueron definidas solo a base de los operadores indicados), con lo que se cumple con la restricción dada.

Pregunta 2

Considerando la estructura lógica relacional como $A = (\mathbb{N}; +; \times)$, se tiene como dominio los naturales y como relaciones ternarias la adición y la multiplicación, se formalizarán los siguientes predicados:

1. El conjunto de todos los triples (m, n, p) tal que $n \neq 0$ y $p = \lfloor \frac{m}{n} \rfloor$ Se define:

$$Cero(x) := \forall y, \times(x, y, x) \wedge +(x, y, y)$$

$$a < b := \exists k, +(a, k, b) \wedge \neg Cero(k)$$

Luego:

$$triples(m, n, p) := \neg Cero(n) \wedge (\exists k, \exists r, r < n, \times(n, p, k) \wedge +(k, r, m))$$

2. El conjunto de los triples (m, n, p) tal que m $\geq$ n y $m \equiv n \mod p$, es decir, $m - n$ es divisible por p. Se define:

$$a \geq b := \exists k, +(b, k, a)$$

$$a|b := \exists x, (\times(a, x, b) \wedge \neg Cero(a))$$

Luego:

$$triples(m, n, p) := \exists k, m \geq n \wedge +(k, n, m) \wedge p|k$$

3. El conjunto de todos los n que son de la forma 2^p, para algún $p \geq 0$

$$Uno(x) := \forall y, \times(x, y, y)$$

$$Dos(x) := \exists y, Uno(y) \wedge +(y, y, x)$$

$$2^p(n) = \exists x, Dos(x) \wedge [Uno(n) \vee (\exists k, \neg Cero(k) \wedge \times(k, x, n) \wedge 2^p(k))]$$

4. El conjunto de todos los pares (p, q) de números primos gemelos. Donde los números primos (p, q) son números primos gemelos si, siendo $q > p$, se cumple $q - p = 2$: Se define:

$$x \neq y := x < y \vee y < x$$

$$Primo(x) := \forall z, (Uno(z) \Rightarrow z|x) \wedge \forall z, (\neg Uno(z) \wedge z \neq x \Rightarrow \neg z|x) \wedge (x|x \wedge \neg Uno(x))$$

Luego:

$$primosGemelos(p, q) := \exists k, Dos(k) \wedge [Primo(p) \wedge Primo(q)] \wedge [p < q \wedge +(k, p, q)]$$

5. El conjunto de los números racionales. Un número racional significa que p y q no deben tener factores en común distintos de 1 y -1: Se define:

$$MenosUno(x) = \exists k, \exists c, Uno(k) \wedge Cero(c) \wedge +(k, x, c)$$

Luego:

$$Racionales(p, q) := \neg Cero(q) \wedge \forall x([\ x|p \wedge x|q\ ] \Rightarrow [Uno(x) \vee MenosUno(x)])$$

Pregunta 3

1. Se espera que todas las listas sean de largo mayor >0. En el caso de que la lista tenga 1 elemento solo tenemos una opción para elegir, si se tienen 2 elementos, como buscamos los elementos disjuntos, elegimos el mayor, si la lista tiene largo mayor a 2 seleccionamos la mayor suma entre la lista sin el ultimo elemento, la lista con el ultimo elemento y la lista que solo contiene al ultimo elemento:

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ \max(a_1, 0) & \text{si } n = 1 \\ \max(F(n-1),\ F(n-2) + a_n) & \text{si } n > 2 \end{cases} \tag{1}$$

Demostración. Para probar la correctitud se procederá por el principio de inducción:

$\boxed{\text{Caso Base:}}$

$n = 0$

$$F(0) = 0$$

En el caso de que se entregue una lista vacía, la máxima satisfacción posible es 0, por lo que $F(0)$ debe devolver 0.

$n = 1$

$$F(1) = \max(a_1, 0)$$

Como la lista solo contiene un valor solo existen 2 sublistas posibles, la lista $[a_n]$ o la lista vacía. Si la lista $[a_n]$ es mayor o igual a 0 entonces se retorna su único valor, en caso contrario se entrega el vacío, en ambos casos se obtiene la sublista con la máxima satisfacción.

$\boxed{\text{Caso Inductivo:}}$

PDQ: $\forall n : F(0) \land F(1) \land \cdots \land F(n-1) \land F(n) \Rightarrow F(n+1)$

$\Rightarrow$ Debemos mostrar que el algoritmo calcula correctamente $F(n{+}1)$. Cuando el algoritmo calcula $F(n+1)$, este establece que:

$$F(n+1) = \max(F(n), F(n-1) + a_{n+1}\ , a_{n+1})$$

Si se tiene una lista de largo n $[a_1, ..., a_n]$ y le agregamos un elemento a_{n+1}, tenemos que F(n) y F(n-1) existen y nos devuelven los respectivos máximos, si queremos agregar a a_{n+1} a las opciones tendremos que sumárselo a F(n-1), de esta forma nos aseguramos que la sublista que se genera solo contiene elementos no adyacentes. En el caso de que agregar el elemento a_{n+1} y quedarnos con la máxima sublista en el conjunto $[a_1, ..., a_{n-1}]$ no nos entregue la máxima suma también se tiene la opción de que la máxima suma sea F(n). Al buscar el máximo entre F(n) y F(n-1)+a_{n+1} nos aseguramos que obtendremos la máxima satisfacción posible de todas las sublistas y que ninguna sublista contiene elementos adyacentes a otros en la lista original.

$\square$

2. Algoritmo que calcula F de forma recursiva:

```
def F(n, lista): #n = largo
    if n == 0:
        return 0
    else if n == 1:
        return max(a[0],0)
    else:
        return max(F(n - 1), F(n - 2) + a[n-1])
```

La complejidad de este algoritmo es $\Theta(2^n)$, como en cada llamada se vuelve a llamar otras 2 veces entonces por cada llamada duplica el tamaño del problema.

3. Algoritmo que calcula F de forma iterativa

```
def F(n, lista):
    Fn_1 = 0
    Fn_2 = 0
    for i in range(n):
        ( Fn_1, Fn_2 ) = ( max(Fn_1, Fn_2+lista[i]),  Fn_1 )

    return Fn_1
```

El algoritmos anterior tiene complejidad $\Theta(n)$, ya que su complejidad esta en el loop, que pasa por todos los elementos de la lista original.