Algunas Aplicaciones de LP

- Diversas son las aplicaciones directas de lógica proposicional.
- A continuación veremos cómo un problema de búsqueda (típico de aplicaciones de Inteligencia Artificial) se puede modelar usando lógica proposicional.
- El problema de las $8\ reinas$ consiste en colocar 8 reinas en un tablero de ajedrez, de tal manera que no haya un par que se ataque mutuamente.
- Este problema se puede generalizar a n reinas.
- Modelaremos el problema para 8 reinas.
- Sea p_{ij} una variable proposicional que es verdadera si existe una reina en la posición (i, j) del tablero.
- Debemos modelar los siguientes hechos.

- Hay una reina por fila:

$$\bigwedge_{j \in A} \bigvee_{i \in A} p_{ij}$$

donde $A = \{1, 2, 3, 4, 5, 7, 8\}$.

- Restricciones:
  - Sólo hay una reina por columna:

$$p_{ij} \rightarrow \bigwedge_{k \in A - \{j\}} \neg p_{ik}, \quad \text{para todo } i, j \in A$$

  - Sólo hay una reina por fila

$$p_{ij} \rightarrow \bigwedge_{m \in A - \{i\}} \neg p_{mj}, \quad \text{para todo } i, j \in A$$

  - Sólo hay una reina por diagonal

$$p_{ij} \rightarrow \bigwedge_{m,k \in A, i-j=m-k, i \neq m} \neg p_{mk}, \quad \text{para todo } i, j \in A$$

- Para el problema de n reinas, basta con redefinir el conjunto A.

- La solución al problema consiste en encontrar una valuación que haga verdadera a todas las fórmulas simultáneamente.

- ¿Cómo podemos hacer esto?

Algoritmos para Encontrar Satisfactibilidad

- La alternativa más ingenua para encontrar una valuación que haga verdadera a una fórmula es usando una tabla de verdad.

- El problema de esto es que, para una fórmula de n variables, deberemos revisar, en promedio 2^{n-1} valuaciones.

- ¿Cuan complejo es el problema?

- Más adelante tendremos algunas respuestas.

- Por el momento veremos dos algoritmos para encontrar satisfactibilidad de fórmulas.

El algoritmo DPLL

- Este es un algoritmo **sistemático** para encontrar satisfactibilidad.
- Propuesto en 1962, utiliza fórmulas en FNC y asigna un valor de verdad de tal manera de simplificar al máximo la fórmula.
- Supongamos que tenemos la siguiente fórmula en FNC:

$$\psi = (A \vee B \vee \neg E) \wedge (B \vee \neg C \vee D) \wedge (\neg A) \wedge (C \vee E)$$

- En el primer paso, el algoritmo se da cuenta que A debe ser falsa.
- Dada esa valuación, el problema se reduce a encontrar el valor de verdad de la siguiente fórmula.

$$\psi(\neg A) = (B \vee \neg E) \wedge (B \vee \neg C \vee D) \wedge (C \vee E)$$

- Se usa la notación $\psi(u)$ para denotar la fórmula que resulta de hacer verdadero el literal u en ψ y luego simplificar.

- Así,

$$\psi(B) = (\neg A) \wedge (C \vee E).$$

- El algoritmo es el siguiente:

```
procedure DPLL(formula CNF: φ)
  Si φ es vacía, retornar 1
  Si hay una cláusula vacía en φ, retornar 0
  Si hay un literal u solo en una cláusula en φ, retornar DPLL(φ(u)
  En caso contrario
     escoja una variable v mencionada en φ
     Si DPLL(φ(v))=1, retornar 1.
     En caso contrario, retornar DPLL(φ(¬ v))
```

- Normalmente en muchos problemas de satisfactibilidad, lo que interesa es además cuál es la valuación que hace verdadera a la fórmula.

- Este algoritmo se puede extender trivialmente para encontrar una valuación.

GSAT

- Una forma alternativa de resolver el problema de satisfactibilidad es a través de algoritmos **estocásticos** (no sistemáticos).

- Estos algoritmos han tenido gran éxito en aplicaciones de inteligencia artificial.

- La idea es partir de una valuación aleatoria e ir modificándola hasta encontrar una valuación precisa.

- Si tenemos "suerte" encontraremos una valuación rápidamente.

- El siguiente es el algoritmo:

```
procedure GSAT(formula CNF: φ, enteros: N_restart, N_flips)
  For i := 1 to N_restarts,
    Escoger una valuación aleatoria A
    For j := 1 to N_flips,
    Si A satisface φ retornar 1
    en caso contrario
      Seleccione la variable v que al cambiar su valor de
        verdad produce el mayor aumento de cláusulas satisfechas;
      Modifique A, cambiando el valor de verdad de v
```

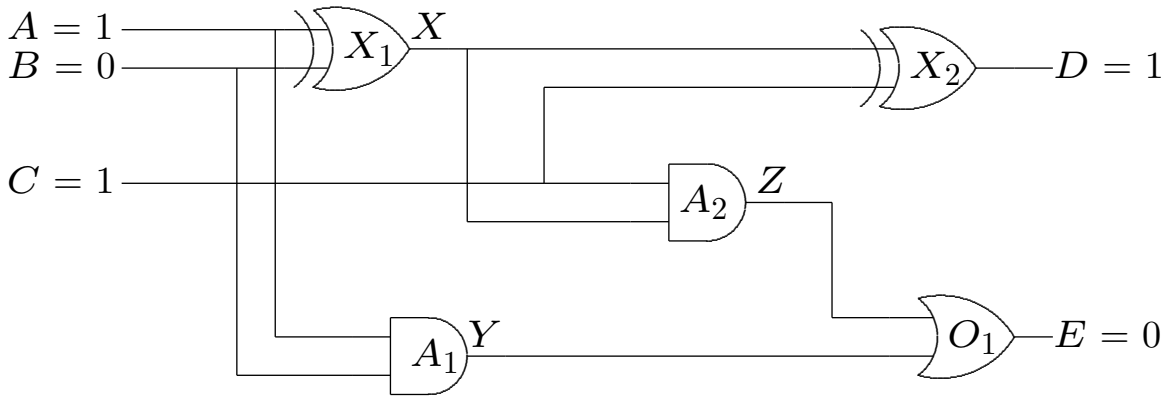
- El algoritmo es *greedy*.

- Es incompleto, pero efectivo.

- Mejoras se han obtenido permitiendo que, con cierta probabilidad decreciente en el tiempo, el algoritmo no elija el mejor *flip*.

Diagnóstico en Circuitos Lógicos

- En la figura se muestra un circuito lógico



  en el cual X_1 y X_2 son compuertas XOR, A_1 y A_2 son compuertas AND y O_1 es una compuerta OR.

- La figura muestra un comportamiento incorrecto del circuito, puesto que la salida debería ser $D = 0$, $E = 1$.

- La pregunta que nos hacemos es:

  ¿cuáles compuertas están fallando?

- Es posible modelar lógicamente el comportamiento del circuito.

- Si una compuerta funciona correctamente, entonces el comportamiento del ésta es el correcto.

- El siguiente conjunto de fórmulas Σ_{Sist} describe el funcionamiento del sistema.

$$
\begin{aligned}
&okX_1 \rightarrow X \leftrightarrow A \otimes B \\
&okA_1 \rightarrow Y \leftrightarrow A \wedge B \\
&okX_2 \rightarrow D \leftrightarrow X \otimes C \\
&okO_1 \rightarrow E \leftrightarrow Z \vee Y \\
&okA_2 \rightarrow Z \leftrightarrow C \wedge X
\end{aligned}
$$

- Las variables proposicionales que comienzan con ok son verdaderas ssi la compuerta en el resto del nombre de la variable funciona correctamente.

- Por otro lado, las observaciones del sistema quedan dadas por las siguientes

fórmulas:

$$A \leftrightarrow \top \qquad B \leftrightarrow \bot \tag{3}$$

$$C \leftrightarrow \top \qquad D \leftrightarrow \top \tag{4}$$

$$E \leftrightarrow \bot \tag{5}$$

- Claramente, si el comportamiento de un sistema no es el correcto y $COMP = \{ok_1, ..., ok_n\}$ son las variables que describen el correcto funcionamiento de las componentes:

  $$\Sigma_{sist} \cup \Sigma_{obs} \cup \{ok_1 \wedge ok_2 \wedge \ldots \wedge ok_n\}$$

  es inconsistente.

- Por otro lado, generalmente lo que interesará encontrar será un diagnóstico, que será un conjunto de componentes que posiblemente están fallando.

- Un diagnóstico Δ es un conjunto de variables tal que $\Delta \subseteq COMP$ y

  $$\Sigma_{sist} \cup \Sigma_{obs} \cup \{\bigwedge_{p \in \Delta} \neg p\} \cup \{\bigwedge_{p \in COMP - \Delta} p\}$$

es consistente.

- Obviamente, lo que nos interesa conocer son los conjuntos Δ con el mínimo conjunto posible de elementos tal que la relación anterior se cumpla. ¿Por qué?

- El conjunto Δ se puede construir incrementalmente a partir de una serie de observaciones.

- Mientras mayor cantidad de observaciones se tenga, menos elementos tendrá el conjunto.

- Pero ¿cómo podemos encontrar un diagnóstico?

- Una estrategia consiste en encontrar conjuntos de $conflictos$.

- Un conflicto es un subconjunto de C de $COMP$ tal que:

$$\Sigma_{sist} \cup \Sigma_{obs} \cup \{ \bigwedge_{p \in C} p \}$$

es inconsistente.

- Si, por ejemplo $\{ok_1, ok_5, ok_6\}$ es un *conjunto conflicto*, entonces significa que las compuertas correspondientes no pueden estar todas juntas funcionando al mismo tiempo.

- Dado un conjunto de observaciones para un sistema que funciona incorrectamente es posible encontrar muchos conjuntos conflicto.

- De estos, algunos serán *minimales* si al quitar un elemento del conjunto, dejan de ser un conjunto conflicto.

- Un diagnóstico es un conjunto que contiene, al menos, un elemento de cada conjunto conflicto que se pueda encontrar. ¿Por qué?

- En nuestro ejemplo, para las observaciones de la figura, es posible encontrar dos conjuntos conflicto minimales. Éstos son: $\{okX_1, okX_2\}$ y $\{okX_1, okA_2, okO_1\}$. Con lo que los diagnósticos son:

$$\{okX_1\}, \{okX_2, okA_2\}, \{okX_2, okO_2\}$$

- La pregunta que ahora queda es ¿cómo podremos encontrar los conjuntos conflicto en forma automática?.

- Necesitamos, al menos, un algoritmo que nos sirva para determinar cuándo un conjunto de fórmulas es inconsistente y que sea eficiente.

OTTER

- Sus características son las siguientes:
  - Puede usar resolución estándar o hiper-resolución .
  - Recibe como entrada a fórmulas proposicionales de cualquier tipo.
  - Para el caso proposicional, es correcto y completo, porque implementa resolución proposicional.

- La entrada común de otter son dos listas:
  - La lista `usable`: las fórmulas de la base de conocimiento.
  - La lista `sos`: contiene las fórmulas queremos obtener como consecuencia de la lista `usable`. Importante: el conjunto de fórmulas debe estar negado.

Sintaxis de OTTER

- Básicamente, la entrada para otter es un archivo con la siguiente estructura:

```
[seteo opciones]

formula_list(usable).
[formulas (cada una terminada en punto)]
end_of_list.

formula_list(sos).
[formulas (cada una terminada en punto)]
end_of_list.
```

- La sintaxis de las fórmulas es similar a la de lógica proposicional. Los conectivos que se ocupan se describen en la siguiente tabla:

negación ($\neg$)	-
disyunción ($\vee$)	|
conjunción ($\wedge$)	&
implicación ($\rightarrow$)	->
equivalencia ($\leftrightarrow$)	<->

- OTTER no es interactivo, por lo que se le debe pasar el archivo como entrada. Ejemplo:

```
otter < archivo_entrada > archivo_salida
```

Ejemplos

- Consideremos el siguiente ejemplo:

```
set(binary_res).

formula_list(usable).
(p & -q) -> (r|s).
-r & -q.
end_of_list.

formula_list(sos).
- (p->s).
end_of_list.
```

- Lo que intenta hacer este ejemplo es demostrar que

$$\{p \wedge \neg q \rightarrow r \vee s, \neg r \wedge \neg q\} \models p \rightarrow s$$

- El siguiente es un resumen de la salida de otter:

```
-------> usable clausifies to:

list(usable).
1 [] -p|q|r|s.
2 [] -r.
3 [] -q.
end_of_list.

-------> sos clausifies to:

list(sos).
4 [] p.
5 [] -s.
end_of_list.

---------------- PROOF ----------------
1 [] -p|q|r|s.
2 [] -r.
3 [] -q.
4 [] p.
5 [] -s.
6 [binary,4.1,1.1,unit_del,3,2,5] $F.
```

```
----------- end of proof ------------
```

- Vemos que se llega a la cláusula vacía `$F`, con lo que hemos demostrado lo que queríamos demostrar.
- Supongamos que queremos saber si `s` se deduce de ese mismo conjunto de cláusulas.
- Deberíamos agregar `-s` a la lista `sos`.
- En este caso OTTER no llega a ninguna respuesta, mostrando el siguiente mensaje dentro de su salida:

```
Search stopped because sos empty.
```

- Por lo tanto, `s` no se deduce del conjunto de cláusulas.
- Es posible usar hiper-resolución con otter reemplazando `set(binary_res)` por `set(hyper_res)`, pero es necesario preocuparse de ubicar las cláusulas con literales positivos en la lista `sos`.

Resolviendo Puzles Lógicos con OTTER

- Es posible utilizar OTTER para resolver puzles lógicos.

- Consideren el siguiente problema:

  En una cierta isla hay individuos de dos clases: aquéllos que siempre dicen la verdad, y aquéllos que siempre mienten. Usted llega a esta isla y se encuentra con tres habitantes A, B y C.
  Le pregunta a A "¿Usted dice la verdad o miente?"
  A balbucea algo que usted no entiende.
  Luego le pregunta a B qué es lo que A dijo.
  B responde, "A dijo que él es un mentiroso".
  C agrega, "No le creas a B, porque miente!".
  ¿Que se puede decir sobre A, B y C?

- El siguiente modelo se puede utilizar para descubrir a el(los) mentiroso(s)...

- El siguiente conjunto de fórmulas modela la situación:

```
formula_list(usable).
a_miente   -> - a_dice_que_miente.
- a_miente -> - a_dice_que_miente.
b_miente <-> a_dice_que_miente.
- c_miente <-> b_miente.
end_of_list.
```

- Nuestro objetivo a continuación debe ser determinar si es que es posible concluir `a_miente` o `b_miente` o `c_miente` o la negación de estos...

- Si agregamos, por ejemplo `c_miente` a la lista de `sos`, obtenemos una demostración:

```
1 [] -a_miente| -a_dice_que_miente.
2 [] a_miente| -a_dice_que_miente.
3 [] b_miente|a_dice_que_miente.
6 [] -c_miente| -b_miente.
7 [] c_miente.
8 [binary,7.1,6.1] -b_miente.
9 [binary,8.1,3.1] a_dice_que_miente.
10 [binary,9.1,2.2] a_miente.
11 [binary,9.1,1.2] -a_miente.
12 [binary,11.1,10.1] $F.
```

- Por lo tanto, C dice la verdad (¿por qué?).

- Siguiendo de manera análoga, podemos concluir que B miente y que nada podemos decir de A.

Logros de los Demostradores Mecánicos de Teoremas

- Usando este demostrador se han resuelto una serie de problemas abiertos en distintas áreas de la matemática (ver `http://www-unix.mcs.anl.gov/AR/otter/`).

- Uno de los más conocidos es el caso de la equivalencia entre las álgebras de Boole y las álgebras de Robbins.

- En 1933, Huntington presentó los siguientes axiomas para las álgebras de Boole:

$$x + y = y + x$$
$$(x + y) + z = x + (y + z)$$
$$(x' + y)' + (x' + y')' = x \qquad (*)$$

  Con la única operación $+$ y $'$

- Herbert Robbins conjeturó que (*) podía ser reemplazada por:

$$(x + y)' + (x' + y)')' = y$$

Sin embargo esta conjetura no pudo ser demostrada hasta 1996, cuando EQP (que es similar a otter) demostró la equivalencia.

- En el año 2000, se publico un artículo en el que, usando Otter, se demuestró que:

$$(((x+y)'+z')+(x+(z'+(z+u)')')')'=z$$

es suficiente para axiomatizar el álgebra de Boole.

Prototype Verification System (PVS)

- PVS es un sistema (desarrollado en SRI) que integra un lenguaje de especificación y un demostrador mecánico de teoremas.

- Las diferencias principales con los demostradores que hemos visto son:
  - Trabaja sobre una lógica de orden superior ($higher\text{-}order\ logic$), aunque soporta lógica proposicional y de primer orden.
  - No usa resolución sino que el cálculo de "secuentes" ($sequents$).
  - Permite una alta interacción con el usuario durante el proceso de demostración.

- Funciona dentro del editor Emacs.

Teorías Proposicionales en PVS

- Las palabras claves `OR`, `AND`, `IMPLIES`, `IFF` y `NOT` representan a los conectivos de nuestro lenguaje proposicional.

- La siguiente es una teoría proposicional en PVS:

```
proposicional: THEORY

BEGIN

A, B, C: bool

ej1: LEMMA
    A IMPLIES (B OR A)

ej2: LEMMA
    ((A IMPLIES (B IMPLIES C)) AND (A IMPLIES B) AND A) IMPLIES C

notlema: LEMMA
    (A IMPLIES B) IMPLIES (B IMPLIES A)
```

```
END proposicional
```

Demostrando teoremas

- Para demostrar un teorema es necesario cargar el demostrador, luego el archivo con la teoría, y luego presionar control+c seguido de p sobre el teorema a demostrar.

- Los dos comandos básicos para demostrar teoremas proposicionales son:
  - `(flatten)` "Desarma" secuentes que tienen conjunciones en el antecedente o disyunciones o implicaciones en el consecuente.
    Por ejemplo, al aplicar `(flatten)` al secuente

$$\frac{A_1 \wedge A_2 \quad A_n}{B \to C}$$

    Se obtiene

$$\frac{A_1 \quad A_2 \quad A_n \quad B}{C}$$

– `(split)` Si es posible, genera sub-objetivos que resultan de desarmar una disyunción en el antecedente o una conjunción en el consecuente del secuente.

Por ejemplo, al aplicar `(split)` al secuente:

$$\frac{A_1 \vee A_2 \quad A_n}{B}$$

Se obtienen dos secuentes (sub-objetivos)

$$\frac{A_1 \quad A_n}{B} \qquad\qquad \frac{A_2 \quad A_n}{B}$$

- El comando `(prop)` realiza estos dos procesos en forma automática hasta que no puede seguir más adelante.