Lógica

Dr. Víctor de la Cueva
vcueva@itesm.mx

Razonamiento

- Es uno de los mecanismos más importantes de la inteligencia humana.
- Su componente central es una Base de Conocimiento (KB).
- Una KB es un conjunto de oraciones o proposición (*sentences*), donde el término "oración" es un término técnico, no es lo mismo que una oración en español u otro lenguaje natural.
- Cada oración es expresada en un lenguaje llamado Lenguaje de Representación del Conocimiento y representa una aserción sobre el mundo.
- Algunas veces llamamos a la oración axioma, cuando la oración es tomada como dada sin ser derivada de otras oraciones.
- Debe haber una forma de agregar (TELL) nuevas oraciones a la KB y una forma de preguntar (ASK) lo que se quiere saber.
- Ambas operaciones involucran inferencia.

Inferencia

- Consiste en el proceso de derivar nuevas oraciones de las viejas.
- Debe responder al requerimiento de que cuando uno hace una pregunta (ASK), la respuesta debe ser derivada (*follow*) de lo que uno le ha dicho (TELL) a la KB previamente.
- La palabra crucial es "derivar".
- La KB inicialmente contiene un conocimiento inicial (background knowledge).
- Tomaremos como ejemplo el "Mundo de Wumpus".

El mundo de wumpus

El **mundo de *wumpus*** es una cueva que está compuesta por habitaciones conectadas mediante pasillos. Escondido en algún lugar de la cueva está el *wumpus*, una bestia que se come a cualquiera que entre en su habitación. El *wumpus* puede ser derribado por la flecha de un agente, y éste sólo dispone de una. Algunas habitaciones contienen hoyos sin fondo que atrapan a aquel que deambula por dichas habitaciones (menos al *wumpus*, que es demasiado grande para caer en ellos). El único premio de vivir en este entorno es la posibilidad de encontrar una pila de oro. Aunque el mundo de *wumpus* pertenece más al ámbito de los juegos por computador, es un entorno perfecto para evaluar los agentes inteligentes. Michael Genesereth fue el primero que lo propuso.

Condiciones

- En el cuadro conteniendo el Wumpus y sus cuadros adyacentes (no diagonales) el agente percibe un HEDOR.
- En los cuadros que contienen un hoyo el agente percibe una BRISA en los cuadros adyacentes.
- En el cuadro que contiene el oro el agente percibe un RESPLANDOR.
- Cuando el agente choca con una pared, se percibe un GOLPE.
- Cuando se mata al wumpus se escucha un GRITO en cualquier lado.
- Las percepciones son dadas al agente en forma de una lista de 5 símbolos
- [Hedor, Brisa, Resplandor, Golpe, Grito] y usar *None* en caso de no tener alguno.
- El agente no puede percibir su posición.

Acciones

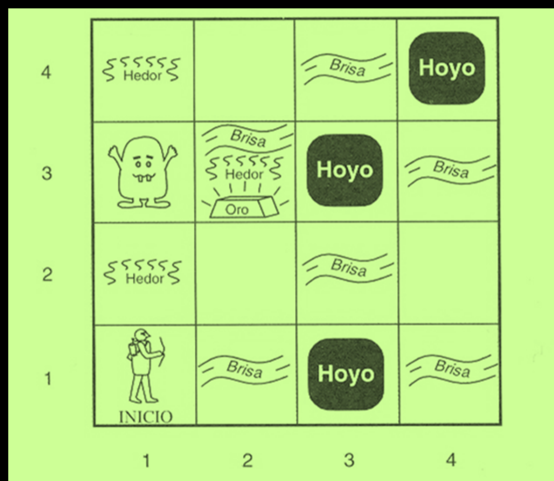
- Hay acciones para ir adelante, girar 90° izquierda o derecha.
- Tomar: recoge un objeto que esté en el mismo.
- Dispara: envía la flecha en la dirección que está el agente y se detiene si pega en el wumpus o choca con la pared.
- El agente sólo tiene 1 flecha.
- Sube: sirve para abandonar la cueva.

Otras condiciones

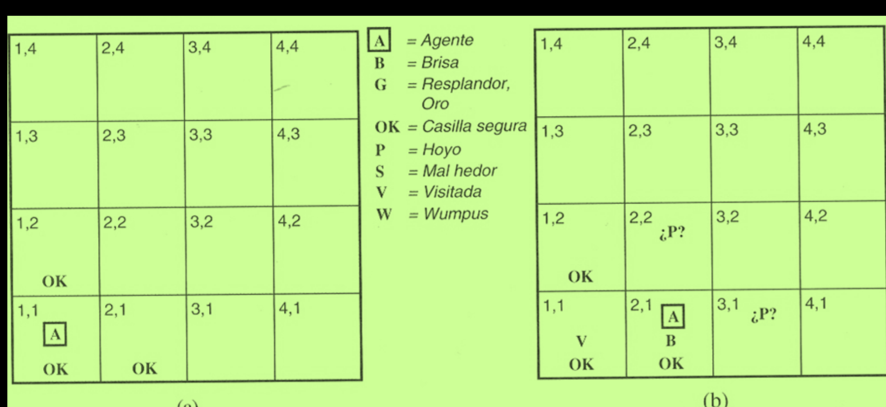
- El agente muere si entra en un cuadro con el wumpus vivo o en un hoyo.
- El agente está a salvo si entra en un lugar con el wumpus muerto (pero huele muy feo).
- El objetivo del agente es encontrar el oro y traerlo al punto de inicio tan rápido como sea posible.
- Rendimiento: +1000 (oro), -10000 (hoyo o comido), -1 (acción), -10 (flecha).
- Entorno: matriz de 4 X 4 con paredes alrededor.
- El agente siempre inicia en [1,1] viendo a la derecha
- Actuadores: adelante (muro), gira 90° derecha o izquierda, agarrar, disparar.

- Sensores: [Hedor, Brisa, Resplandor, Golpe, Grito] e.g. [H,B,N,N,N]
- La posición del wumpus y del oro se escogen en forma aleatoria con probabilidad uniforme sobre los cuadros excepto el de inicio.
- Cada cuadro diferente del inicio puede tener un hoyo con probabilidad de 0.2 (el oro puede estar en un hoyo).
- El agente puede optar por salir sin oro. En ese caso se generará otro mundo para que entre pero no se le penaliza por morir.
- El 21% de las veces el oro cae en un hoyo y no puede ser alcanzado por el agente.
- El agente debe combinar conocimiento actual con el ganado en tiempos anteriores.

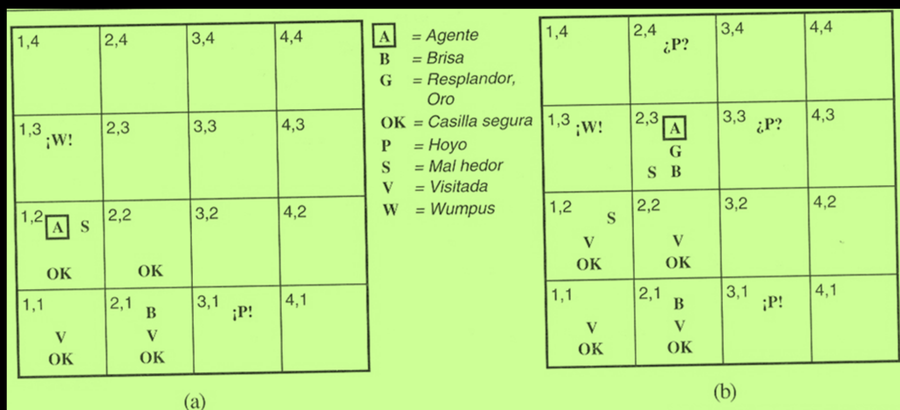
Un estado ejemplo



Avance 1



Avance 2



(a)

(b)

Variantes

- Múltiples agentes cooperantes
- El wumpus se mueve
- Múltiples lugares con oro cada uno con distinto valor
- Múltiples wumpus

Representación y razonamiento

- En todos los casos un agente debe ser capaz de representar conocimiento como:

“Hay un hoyo en [2,2] o en [3,1]”

“No hay un wumpus en [2,2]”

- Con ese conocimiento debe ser capaz de realizar inferencias (razonar).
- Una buena herramienta para resolver este problema y poder razonar sobre él (hacer inferencia en base al conocimiento previo para generar nuevo) es la Lógica.

Lógica

- Una lógica es un sistema de razonamiento que consta de dos partes fundamentales:
  - Un lenguaje de Representación del Conocimiento, el cual, como todo lenguaje, debe tener una SINTAXIS y una SEMÁNTICA, como todo lenguaje.
  - Un método de inferencia.
- Hay muchas lógicas pero las más comunes son dos:
  - Lógica Proposicional.
  - Lógica de Predicados.
- La lógica proposicional tiene pocas aplicaciones pero es muy didáctica. Daremos un breve repaso de la misma debido a que este tema ya lo cubrieron.
- La lógica de predicados, en la que se basa Prolog, es muy utilizada.

Sintaxis de la Lógica Proposicional

Sentencia → *Sentencia Atómica* | *Sentencia Compleja*
Sentencia Atómica → **Verdadero** | **Falso** | *Símbolo Proposicional*
Símbolo Proposicional → **P** | **Q** | **R** | **...**
Sentencia Compleja → (Sentencia)
| Sentencia Conectiva Sentencia
| ¬ Sentencia

Conectiva → $\wedge$ | $\vee$ | $\Leftrightarrow$ | $\Rightarrow$

Ejemplo: $((\neg P) \vee (Q \wedge R)) \Rightarrow S$

Semántica de la lógica proposicional

- La semántica de la LP se da por medio de tablas de verdad.

P	Q	~P	P∧Q	P∨Q	P→Q	P↔Q
F	F	V	F	F	V	V
F	V	V	F	V	V	F
V	F	F	F	V	F	F
V	V	F	V	V	V	V

Una pequeña KB

- Hecho sobre un agujero en el primer cuadro: No hay un hoyo en [1,1]

  $R_1 : \sim P_{11}$

- Un cuadrito tiene brisa si alguno de sus cuadros 4-adyacentes tiene un hoyo. Los cuadros relevantes por ahora son [1,1], ni en [2,1]:

  $R_2: B_{11} \leftrightarrow (P_{12} \vee P_{21})$

  $R_3: B_{21} \leftrightarrow (P_{11} \vee P_{22} \vee P_{31})$

- Hechos sobre la brisa en los primeros dos cuadros: No hay brisa en [1,1], y sí en [2,1]

  $R_4: \sim B_{11}$

  $R_5: B_{21}$

- Queremos probar que no hay un hoyo en [1,2]: $\sim P_{12}$.

Procedimientos de inferencia

- Hay al menos tres procedimientos de inferencia que se pueden aplicar en el caso de la lógica proposicional:
  - Tabla de verdad (enumeración completa)
  - Razonamiento directo (usando reglas de inferencia)
  - Resolución (sólo usa una regla de inferencia)
- Nuestra meta es que, dada una oración α y una KB, decidir si la oración puede ser derivada o inferida (*follows logically*) a partir de las oraciones de la KB, que ya se saben que son válidas, es decir: $KB \vdash \alpha$, a lo que se le conoce como *logical entailment*.
- Se dice que $KB \vdash \alpha$ si $KB \rightarrow \alpha$ es una tautología, es decir, si cada vez que KB sea verdadera, α también es verdadera. La única forma en la que KB es verdadera es cuando todas as oraciones β_i que la forma lo son, es decir: $KB = \beta 1 \wedge \beta 2 \wedge \ldots \wedge \beta n$.

Las dos primeras

- Inferencia por tabla de verdad es una implementación directa de la definición de *entailment*:
  - Se hace una tabla de verdad para todas las variables y se prueba que KB $\rightarrow \alpha$ es una tautología.
  - Que es lo mismo que probar que siempre que KB es V (la conjunción de todas sus oraciones debe ser verdadera), α también lo es.
  - Requiere una gran cantidad de tiempo y memoria.
- Inferencia por razonamiento directo (demostración):
  - Aplica las reglas de inferencia a las oraciones de la KB (ya probadas) y va agregando los resultados a la KB, hasta derivar α.
  - Si ya no se puede aplicar alguna regla a las oraciones de la KB de tal forma que dé como resultado una oración que no se encuentra en la KB, el proceso termina diciendo que α no se puede derivar de KB.
  - Requiere mucho poder de cómputo (un cerebro humano).

Conceptos relacionados con *entailment*

- Equivalencia lógica: dos oraciones α y β son lógicamente equivalentes si son verdaderas en el mismo conjunto de modelos, es decir, si tienen la misma tabla de verdad, es decir, si $\alpha \leftrightarrow \beta$ es una tautología. Se escribe $\alpha \equiv \beta$(e.g. $P \rightarrow Q \equiv \sim P \vee Q$).
- En una oración, cualquier cosa puede sustituirse por su equivalencia.
- Validez: Una oración es válida si es verdadera para todos sus modelos. También se le conoce como tautología (e.g. $Q \vee \sim Q$).
  - Teorema de deducción: Para dos oraciones dadas α y β, $\alpha \vdash \beta$ si y sólo si la oración $\alpha \rightarrow \beta$ es válida.
- Satisfacibilidad (*satisfiability*): una oración es satisfacible si es verdadera en algún modelo.
  - El problema de determinara la satisfacibilidad de una oración en lógica proposicional – el problema SAT – fue el primer problema que se probó ser NP-completo.

Validez y satisfacibilidad

- Validez y satisfacibilidad están, desde luego, conectados:
  - α es válida si y sólo si $\sim\alpha$ es insatisfacible.
- También, tenemos el siguiente resultado útil:
  - $\alpha \vdash \beta$ si y sólo si la oración $(\alpha \wedge \sim\beta)$ es insatisfacible.
- De acuerdo al resultado anterior, para probar que $KB \vdash \alpha$ o sea, que $KB \rightarrow \alpha$, es decir $(\sim KB \vee \alpha)$, por medio de la insatisfacibilidad, es decir que $(KB \wedge \sim\alpha)$, corresponde exactamente a la prueba matemática por reducción al absurdo, también llamada refutación o contradicción.
- Para esta prueba basta suponer que $\sim\alpha$ es verdadera, meterla a la KB y hacer derivaciones aplicando reglas de inferencia hasta obtener un absurdo (e.g. $Q \wedge \sim Q$), en ese momento podemos decir que $\sim\alpha$ no puede ser verdadera y por lo tanto α es verdadera.

Reglas de inferencia

- Modus Ponens $\dfrac{\alpha \Rightarrow \beta, \alpha}{\beta}$
- Y-Eliminación $\dfrac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\alpha_i}$
- Y-Introducción $\dfrac{\alpha_1, \alpha_2, \ldots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}$
- O-Introducción $\dfrac{\alpha_i}{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n}$
- Doble Negación $\dfrac{\neg\neg\alpha}{\alpha}$

Resolución

- También es una regla de inferencia.
  - Resolución unitaria
  - Resolución

o

$$\frac{\alpha \vee \beta, \neg\beta}{\alpha}$$

$$\frac{\alpha \vee \beta, \neg\beta \vee \gamma}{\alpha \vee \gamma}$$

$$\frac{\alpha \Rightarrow \beta, \neg\beta \Rightarrow \gamma}{\alpha \Rightarrow \gamma}$$

Resolvente

Resolución General

- Resolución unitaria, donde α_i y β son complementarios:

$$\frac{\alpha_1 \vee \ldots \vee \alpha_k, \beta}{\alpha_1 \vee \ldots \vee \alpha_{i-1} \vee \alpha_{i+1} \vee \ldots \vee \alpha_k}$$

- Resolución general, donde α_i y β_j son complementarios:

$$\frac{\alpha_1 \vee \ldots \vee \alpha_k, \beta_1 \vee \ldots \beta_n}{\alpha_1 \vee \ldots \vee \alpha_{i-1} \vee \alpha_{i+1} \vee \ldots \vee \alpha_k \vee \beta_1 \vee \ldots \vee \beta_{j-1} \vee \beta_{j+1} \vee \ldots \vee \beta_k}$$

Equivalencias

$(\alpha \wedge \beta)$	$\equiv$	$(\beta \wedge \alpha)$	Conmutatividad de $\wedge$
$(\alpha \vee \beta)$	$\equiv$	$(\beta \vee \alpha)$	Conmutatividad de $\vee$
$((\alpha \wedge \beta) \wedge \gamma)$	$\equiv$	$(\alpha \wedge (\beta \wedge \gamma))$	Asociatividad de $\wedge$
$((\alpha \vee \beta) \vee \gamma)$	$\equiv$	$(\alpha \vee (\beta \vee \gamma))$	Asociatividad de $\vee$
$\neg(\neg\alpha)$	$\equiv$	α	Eliminación de la doble negación
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\beta \Rightarrow \neg\alpha)$	Contraposición
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\alpha \vee \beta)$	Eliminación de la implicación
$(\alpha \Leftrightarrow \beta)$	$\equiv$	$((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	Eliminación de la bicondicional
$\neg(\alpha \wedge \beta)$	$\equiv$	$(\neg\alpha \vee \neg\beta)$	Ley de Morgan
$\neg(\alpha \vee \beta)$	$\equiv$	$(\neg\alpha \wedge \neg\beta)$	Ley de Morgan
$(\alpha \wedge (\beta \vee \gamma))$	$\equiv$	$((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	Distribución de $\wedge$ respecto a $\vee$
$(\alpha \vee (\beta \wedge \gamma))$	$\equiv$	$((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	Distribución de $\vee$ respecto a $\wedge$

Ejemplo en el MW demostrar: $\sim P_{12}$

- Aplicando eliminación bicondicional a R2:

$(B11 \rightarrow (P12 \vee P21)) \wedge ((P12 \vee P21) \rightarrow B11)$

- Aplicando Y-eliminación a R6:

$((P12 \vee P21) \rightarrow B11)$

- Aplicando la equivalencia lógica contrapositiva a R7:

$(\sim B11 \rightarrow \sim(P12 \vee P21))$

- Modus Ponens con R8 y R4:

$\sim(P12 \vee P21)$

- De Morgan a R9:

$\sim P12 \wedge P21$

- Y-eliminación a R10: $\sim P12$.

Con la computadora

- A la derivación anterior se le conoce como prueba.
- La prueba anterior la hicimos a mano (usando el cerebro).
- Podemos aplicar algún método de búsqueda (BFS, DFS, etc) para encontrar una secuencia de pasos que constituyan una prueba. Sólo se requiere definir el problema como sigue:
  - Estado inicial: La KB inicial.
  - Acciones (para derivar hijos del nodo actual): todas las reglas de inferencia aplicadas a TODAS las oraciones en las que se pueda aplicar (que hagan match con la parte superior de la regla).
  - Resultado: La oración resultante (en la parte baja de la regla) se agrega a la KB.
  - Meta: La meta es el estado que contiene la oración que estamos tratando de probar.
- Este método resulta ser muy poco eficiente porque a veces aplicamos una regla cuyo resultado no sirve para nada. Se requiere decidir qué regla aplicar (experiencia).
- Una forma de no tener que decidir qué regla aplicar es tener sólo una regla para aplicar.

Prueba por Resolución

- Las reglas de inferencia vistas son seguras (*sound*) pero la completez (*completeness*) del algoritmo de búsqueda depende del conjunto de reglas que se tenga disponible (debe ser adecuado).
- ¿Cómo sabemos si el conjunto es adecuado? (e.g. si en el ejemplo se quita la eliminación de la bicondicional, no se puede hacer la prueba).
- La prueba por resolución requiere una sola regla de inferencia, resolución, y proporciona un algoritmo de inferencia completo cuando se combina con un algoritmo de búsqueda completo.
- Desde luego que tiene su precio, la resolución sólo se aplica a cláusulas, es decir, disyunción de literales (literales positivas o negadas, unidas por OR), por lo que toda la KB debe estar formada sólo por cláusulas.

Forma Normal Conjuntiva (CNF)

- Afortunadamente, cualquier oración se puede transformar a una conjunción de cláusulas y de esta forma, usando la Y-eliminación, meter cada una de las cláusulas a la base de conocimientos.
- A esta conjunción de disyunciones se le conoce como Forma Normal Conjuntiva y el procedimiento para obtenerla es muy simple. Dada una oración:
  1. Eliminar todas las $\leftrightarrow$ usando la eliminación de la bicondicional (o doble implicación).
  2. Eliminar todas las $\rightarrow$ usando la eliminación de la implicación.
  3. Aplicar doble negación y De Morgan para dejar negaciones ~ sólo en literales.
  4. Aplicar la Ley Distributiva de $\vee$ sobre $\wedge$, siempre que sea posible.
- Ejemplo, convertir toda la KB del MW a CNF.

Algoritmo de resolución

- Usa el principio de contradicción, es decir, para mostrar que KB$\vdash\alpha$, se demuestra que (KB $\wedge$ ~α) es insatisfacible:
  - Convertir (KB $\wedge$ ~α) a CNF.
  - Aplicar resolución a las cláusulas resultantes.
    - Cada para que contiene literales complementarias es resuelto para producir una nueva cláusula, la cual se agrega a la KB si es que no está ya incluida.
  - El proceso continua hasta que pasa una de dos cosas:
    - No hay nuevas cláusulas que pueda ser agregadas, en cuyo caso KB no deriva a α (KB does not entail α), o
    - Dos cláusulas se resuelven para obtener la cláusula vacía (representada por □) en cuyo caso KB deriva a α (α es verdadera).
  - La cláusula vacía (una disyunción de ninguna variable) es equivalente a falso (falacia) porque una disyunción es verdadera si al menos una de sus variables es verdadera.
  - Otra forma de verlo es que la cláusula vacía representa una contradicción porque sólo se obtiene de resolver dos variables complementarias P y ~P, que deben estar en la KB, haciéndola inconsistente (no válida).

Ejemplo: Probar ~P12 por resolución

¡ El algoritmo de resolución es completo!

Cláusulas de Horn y cláusulas definidas

- Algunas KB del mundo real satisfacen ciertas restricciones en cuanto a la forma de las oraciones que contiene, lo cual permite usar un algoritmo más restrictivo pero más eficiente.
- Una de tales restricciones es la Cláusula Definida, la cual es una disyunción de literales de las cuales exactamente una es positiva (e.g. $\sim L11 \lor \sim Brisa \lor B11$).
- Ligeramente más general es la Cláusula de Horn (Alfred Horn, 1951), la cual es una disyunción de literales de las cuales a lo más una es positiva.
  - Todas las cláusulas definidas son cláusulas de Horn.
  - También lo son las cláusulas sin literales positivas, las cuales son llamadas cláusulas meta.
  - Las cláusulas de Horn son cerradas con respecto a la resolución: si se resuelven dos cláusulas de Horn se obtiene una cláusula de Horn.

KB conteniendo sólo cláusulas de Horn

- Son interesantes por tres razones:
  1. Cada cláusula definitiva puede ser escrita como una implicación en la que la premisa es una conjunción de literales positivas y cuya conclusión es una sola literal positiva:
     - Por ejemplo: $(\sim L11 \lor \sim Brisa \lor B11)$ puede ser escrita como la implicación $(L11 \land Brisa) \to B11$ (con la regla de eliminación de la implicación), lo que significa en el MW que si el agente está en [1,1] y siente brisa, entonces el cuadro [1,1] tiene brisa.
     - En una cláusula de Horn, la premisa es llamada *body*, y la conclusión es llamada *head*.
     - Una oración consistente de una sola literal positiva, tal como L11, se llama hecho (*fact*), el cual también puede ser escrito como implicación *True* $\to$ L11, pero es más simple escribir L11.
  2. La inferencia con cláusulas de Horn puede ser hecha por medio de los algoritmos de encadenamiento hacia adelante (*forward-chaining*) y encadenamiento hacia atrás (*backward-chaining*).
     - Estos algoritmos son muy naturales en el sentido de que los pasos de inferencia son obvios y fáciles de seguir para los humanos.
     - Este tipo de inferencia en la base de la Programación Lógica.
  3. La decisión de derivación (*entailment*) con cláusulas de Horn puede ser hecha en *tiempo lineal* respecto al tamaño de la KB.

Encadenamiento hacia adelante

- El algoritmo *forward-chaining*, representado por PL-FC-ENTAILS?(KB,Q) determina si una proposición de un solo símbolo q – la consulta (*the query*) – es derivada (*entailed*) por la KB de cláusulas de Horn.
  - Inicia con los hechos conocidos (literales positivas) en la KB.
  - Si todas las premisas de una implicación son conocidas su conclusión es agregada al conjunto de hechos conocidos.
  - Este proceso continua hasta que la consulta q es agregada o hasta que no se puede hacer ninguna inferencia.
- El punto principal a recordad es que corre en tiempo lineal.
- Es seguro: cada inferencia en básicamente una aplicación del Modus Ponens.
- Es completo: cada oración atómica que se pueda derivar (*entailed*) va a ser derivada
- Es un ejemplo del concepto más general de razonamiento dirigido por datos (*data-driven reasoning*), en el cual el foco de atención inicia en los datos conocidos.

Encadenamiento hacia atrás

- El algoritmo *backward-chaining*, como su nombre lo indica, trabaja hacia atrás a partir de la consulta.
  - Si se sabe que la consulta es verdadera entonces ya no se requiere ningún trabajo.
  - De otra forma, el algoritmo encuentra las implicaciones en la KB en las cuales la conclusión es q.
  - Si se puede probar (con *backward-chaining*) que todas las premisas de una de esas implicaciones son verdaderas, entonces q es verdadera.
- El algoritmo es esencialmente idéntico al algoritmo de búsqueda en grafos Y-O (AND-OR-GRAPH-SEARCH).
- Una implementación eficiente corre en tiempo lineal.
- *Backward-chaining* es una forma de razonamiento dirigido por metas (*goal-directed reasoning*).

Ejemplo con un grafo And-Or

$P \rightarrow Q$

$L \wedge M \rightarrow P$

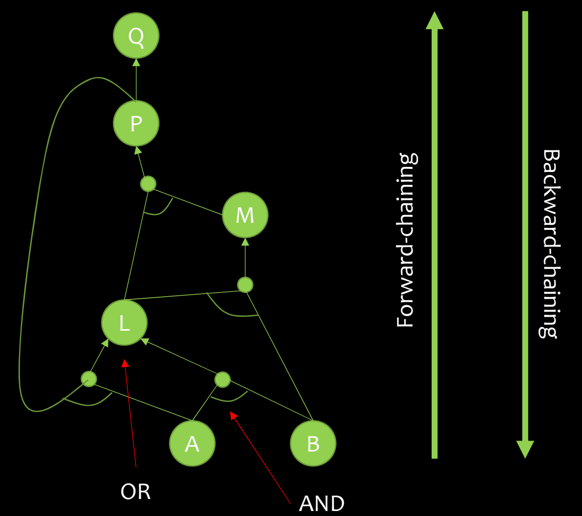
$B \wedge L \rightarrow M$

$A \wedge P \rightarrow L$

$A \wedge B \rightarrow L$

A

B



Memoization

- *Forward-chaining* en problemas de búsqueda en grafos es un ejemplo de programación dinámica, donde las soluciones a los subproblemas son construidas incrementalmente a partir de los más pequeños y usando cache para evitar recalcular.
- Se puede obtener un efecto similar con *Backward-chaining* usando memoization, esto es, haces cache de las soluciones de sub-metas conforme se van encontrando y se reusan cuando la submeta vuelve a ocurrir, en lugar de repetir el cálculo.

Inferencia basada en Model-Checking

- Hay dos familias de algoritmos eficientes para hacer inferencia proposicional basada en verificación del modelo (*Model-Checking*):
  - Un enfoque basado en búsqueda *Backtracking* (basada en caminos).
  - Un enfoque basado en búsqueda *Hill-Climbing* (búsqueda local).
- Estos algoritmos son parte de la "tecnología" de la lógica proposicional.
- Son para verificar satisfacibilidad: el problema SAT.
- Es clara la conexión entre encontrar un modelo que satisfaga una oración lógica y encontrar una solución al problema de satisfacción de restricciones (CSP).
- Son muy importantes porque muchos problemas combinatorios en ciencias computacionales pueden reducirse a verificación de la satisfacibildad de una oración.

Búsqueda Backtracking para CSPs

- Búsqueda Backtracking (BTS) funciona sobre asignaciones parciales mientras que Búsqueda Local funciona sobre asignaciones completas.
- Un problema es conmutativo si el orden de aplicación de cualquier conjunto de acciones dado no afecta a la salida.
- CSPs son conmutativos porque cuando se asignan valores a las variables, alcanzamos la misma asignación parcial sin importar el orden.
- De esta forma, podemos considerar sólo una variable en cada nodo en el árbol de búsqueda.

Búsqueda Backtracking (BTS)

- El término es usado para Búsqueda en Profundidad (*Depth-First Search*) que escoge valores para una variable a un tiempo y regresa cuando una variable no tiene valores legales para asignar.
- BTS repetidamente selecciona una variable no asignada y entonces prueba todos los valores en el dominio de la variable en turno, tratando de encontrar una solución.
- Si se encuentra una inconsistencia BACKTRACK regresa falla, causando que la llamada previa trate de probar otro valor.

Sintaxis Lógica de Primer Orden (FOL)

Sentencia	$\rightarrow$	*SentenciaAtómica*
	$\mid$	(*Sentencia Conectiva Sentencia*)
	$\mid$	*Cuantificador Variable*... *Sentencia*
	$\mid$	¬*Sentencia*
SentenciaAtómica	$\rightarrow$	*Predicado*(*Término*...) | *Término* = *Término*
Término	$\rightarrow$	*Función*(*Término*...)
	$\mid$	*Constante*
	$\mid$	*Variable*
Conectiva	$\rightarrow$	$\Rightarrow \mid \wedge \mid \vee \mid \Leftrightarrow$
Cuantificador	$\rightarrow$	$\forall \mid \exists$
Constante	$\rightarrow$	$A \mid X_1 \mid$ *Juan* $\mid \cdots$
Variable	$\rightarrow$	$a \mid x \mid s \mid \cdots$
Predicado	$\rightarrow$	*AntesDe* | *TieneColor* | *EstáLLoviendo* | ⋯
Función	$\rightarrow$	*Madre* | *PiernaIzquierda* | ⋯

Diferencias principales

- Usa predicados, los cuales deben llevar cuantificadores para ser proposiciones (oraciones).
- Los cuantificadores son dos:
  - Universal (∀)
  - Existencial (∃)
- El algoritmos de CNF debe quitar primero los cuantificadores, lo cual requiere utilizar particularización universal y existencial.
- Debe correr un algoritmo llamado Unificación, que incluye sustituciones (como las alpha-reducciones del cálculo lambda).
- Esto complica un poco el proceso pero sigue haciendo básicamente los mismos pasos.

Programación Lógica

- Es una tecnología que está muy cercana a la forma declarativa ideal que se presentó en la lógica:
  - Los sistemas deben ser construidos para expresar conocimiento en un lenguaje formal.
  - Los problemas deben resolverse corriendo un proceso de inferencia sobre dicho conocimiento.
  - El ideal es resumido en la ecuación de Robert Kowalski: Algoritmos = Lógica + Control.

Prolog

- Es el lenguaje de programación lógica mayormente usado.
- Es usado principalmente como:
  - Lenguaje de prototipado rápido
  - Tareas de manipulación simbólica como la escritura de compiladores (Van Roy, 1990)
  - Parseo de leguaje natural (Pereira y Warren, 1980)
  - Muchos sistemas expertos para leyes, medicina, finanzas y otros dominios, fueron escritos en Prolog.

Programa en Prolog

- Los programas en Prolog son conjuntos de cláusulas de Horn escritas en una notación un poco diferente a la de la LPO (FOL):
  - Mayúsculas para variables y minúsculas para constantes (opuesto a la lógica)
  - Las comas separan las conjunciones en una cláusula
  - Las cláusulas son escritas al revés: $A \wedge B \rightarrow C$, se escribe C :- A, B
  - La notación [E|L] denota una lista con E como su primer elemento y L como el resto (car y cdr de funcional)
- La ejecución de los programas en Prolog se hace por medio del algoritmo de búsqueda en profundidad con encadenamiento hacia atrás (*depth-first backward-chaining*), donde las cláusulas son probadas en el orden en el cual se escriben en la KB.

Aspectos de Prolog

- Algunos aspectos de Prolog caen fuera de la inferencia lógica estándar:
  - Hay un conjunto de funciones aritméticas pre-construidas (*built-in*). Las literales que usan esos símbolos de funciones son "probadas" mediante la ejecución de código más que haciendo inferencia. Por ejemplo, la meta X is 4 + 3 es exitosa con X restringido a 7, mientras que la meta 5 is X + Y falla debido a que la función no hace resolución arbitraria de ecuaciones.
  - Tiene predicados pre-construidos que causan efectos secundarios cuando se ejecutan. Estos incluyen predicados input/output y assert/retract para modificar la KB. Tales predicados no tienen contraparte en lógica y pueden producir resultados confusos, por ejemplo, si los hechos son *asserted* en la rama de un árbol de prueba que eventualmente falla.
  - La prueba de ocurrencia (*occur check*) es omitida en el algoritmo de unificación de prolog. Esto significa que se pueden realizar algunas inferencias inseguras (*unsound*); casi no se hacen en un problema en la práctica.
  - Usa búsqueda *depth-first backward-chaining* sin chequeos para recursión infinita. Esto lo hace muy rápido cuando se le da el conjunto de axiomas adecuado, pero es incompleto cuando se da alguno equivocado.

Referencias

- S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. 3rd ed, Pearson (2010).