Ignacio Méndez Pérez
2023

Resumen Inteligencia Artificial

ASP (*Answer Set Programming*): Semántica de programas sin variables

Una regla en programación lógica es un objeto de la forma:

$$Head \leftarrow Body$$

donde $Head$ y $Body$ son conjuntos de átomos.

Ejemplo: $\{p, q\} \leftarrow \{r, s\},\ \{t\} \leftarrow \{\},\{\} \leftarrow r$

Clingo siempre tratará de buscar un modelo satisfacible en donde la menor cantidad de expresiones sean verdaderas. Por lo que la disyunción del lado izquierdo entregará solo 1 como verdadero, a menos que otra expresiones fuercen a ambas ser verdaderas.

En el sistema ***clingo*** estas reglas, respectivamente se anotan así
Estas reglas significan, respectivamente:

```
1 p;q :- r,s.
2 t.
3 :- r.
```

1) Si las condiciones de `r` y `s` son verdaderas, entonces al menos una de las 2 variables `p` y `q` deben ser verdaderas.

2) Simplemente afirma que `t` es verdadero.

3) Si `r` es verdadero, entonces el modelo es insatisfacible.

Definición: Un <u>programa</u> es un conjunto de reglas.

- Una regla $Head \leftarrow Body$ intuitivamente establece que si $Body$ es parte de un modelo, entonces al menos algo en $Head$ también debe ser verdadero.

Definición: Una <u>regla</u> en programación en lógica es un objeto de la forma:

$$Head \leftarrow Pos, not(Neg)$$

Ejemplo: $\{p, q\} \leftarrow \{r, s\},\ not(\{t\}),\{t\} \leftarrow \{\},not(\{\}),\{\} \leftarrow not(\{r, s\})$

En el sistema clingo estas reglas, respectivamente se anotan así
Estas reglas significan, respectivamente:

```
1 p;q :- r,s,not t.
2 t.
3 :- not r, not s.
```

1) Si se cumple que `r` y `s` son verdaderos, y que `t` es falso, entonces al menos una de las 2 variables `p` y `q` deben ser verdaderas.
2) Simplemente afirma que `t` es verdadero.
3) Si se cumple alguna de las 2 condiciones, (`not r`) o (`not s`), lo que es equivalente a decir, si es que alguna de las variables `r` y `s` es falsa, entonces el modelo será insatisfacible.

Podemos simular un OR usando 2 reglas:

```
iluminado :- ampolleta_encendida.
iluminado :- de_dia, ventana_abierta.

ampolleta_encendida, de_dia, ventana_abierta.
```

Cada una de las expresiones las denominamos “átomos”

Me indica que al menos 1 de las variables que contiene es verdadera.

Variables en clingo

- Representaremos las variables por letras **mayúsculas** o por palabras que comiencen con letra mayúscula.
- `guilty(nombre)` será una instancia de variable. En este caso se usarán letras **minúsculas**.

```
innocent(Suspect) :- motive(Suspect), not guilty(Suspect).

motive(harry).
motive(sally).
guilty(harry).
```

Un ejemplo más complejo: Hay 2 oficinas. Cada una tiene 2 ampolletas y una ventana. La oficina estará iluminada en caso de que la ampolleta esté encendida o la ventana esté abierta:

```
oficina(o1).
oficina(o2).
ampolleta(a11).
ampolleta(a12).
ampolleta(a21).
ampolleta(a22).
instalada_en(a11,o1).
instalada_en(a12,o1).
instalada_en(a21,o2).
instalada_en(a22,o2).
conectado_con(i11,a11).
conectado_con(i12,a12).
conectado_con(i21,a21).
conectado_con(i22, a22).
ventana(w1).
ventana(w2).
en(w1,o1).
en(w2,o2).

emite_luz (A) :- interruptor_encendido(I) , conectado_con(I,A).
iluminada (O) :- emite_luz(A) , instalada_en(A, O).
iluminada (O) :- ventana(W), en(W,O), abierta(W).

interruptor_encendido(i11). % Con esto se ilumina la oficina 1
abierta(w2).                % Con esto se ilumina la oficina 2
```

Ignacio Méndez Pérez
2023

La siguiente expresión indica una disyunción de átomos.

```
1{l ; p ; q }2.
```

Como vimos anteriormente, clingo siempre calculara los modelos posibles con la menor cantidad de átomos verdaderos posibles. Sin embargo, podemos especificar la cantidad mínima (1) y máxima (2) de átomos que queramos que sean verdaderas utilizando las llaves {} y a cada extremo la cantidad mínima y máxima de átomos que queremos que sean verdaderos para nuestro modelo.

Búsqueda Link útil para visualizar algoritmos de búsqueda

- En búsqueda informada, usamos una función de estimación del costo de un nodo del árbol de búsqueda a una solución. La denotamos como:

$$h(n)$$

- En el problema de navegación, si:

$$\Delta x = |x_{obj} - x|, \quad \Delta y = |y_{obj} - y|$$

donde (x, y) es la posición actual y (x_{obj}, y_{obj}) es la posición del objetivo.
La siguiente es una posible heurística (también llamada distancia *octile*).

$$h(x, y) = |\Delta x - \Delta y| + \sqrt{2}\min\{\Delta x, \Delta y\}$$

- Como vimos en el ejemplo, usar h conduce a soluciones no óptimas.

- Es posible encontrar soluciones óptimas al incorporar el **costo** incurrido hasta llega a un nodo n.

- Denotamos este costo como $g(n)$.

- Luego, podemos ordenar la frontera de búsqueda por la siguiente función:

$$f(n) = g(n) + h(n)$$

Pasos Algoritmo A^*

1. **Inicialización**:

- Inicializa `open_list` como una cola de prioridad vacía.
- Inicializa `closed_list` como un conjunto vacío.
- Coloca el nodo de inicio en la cola de prioridad `open_list` con un costo real (g) de `0` y un camino vacío hasta el momento. Inicializa un conjunto de nodos visitados como vacío.

2. **Bucle principal**: Mientras `open_list` no esté vacía, realiza lo siguiente:

    a. Extrae el nodo con el menor costo total (`g(n) + h(n)`) de `open_list`.
    b. Si el nodo extraído es el objetivo, hemos encontrado el camino más corto. Devuelve el camino hasta este nodo.
    c. Si el nodo ha sido visitado previamente (está en `closed_list`), pasamos al siguiente nodo (siguiente iteración).
    d. De lo contrario, realizamos los siguientes pasos:
        i. Lo marca como visitado (lo agrega a `closed_list`).
        ii. Para cada **vecino no visitado** del nodo actual:
            - Calcular el costo real (`g`) desde el nodo de inicio hasta el vecino a través del nodo actual.
            - Calcular el costo estimado (`h`) desde el vecino hasta el objetivo utilizando la función heurística.
            - Calcular el costo total (`f`) del vecino como la suma de `g(n)` y `h(n)`.
            - Agregar el vecino a la `open_list` con su costo total (`f`) como prioridad y el camino hasta el vecino incluyendo el nodo actual.

3. Si `open_list` se vacía y no se encontró el objetivo, significa que no hay un camino desde el nodo de inicio hasta el nodo objetivo. Devuelve un resultado nulo o un mensaje indicando que no se encontró un camino.

```python
def a_star_search(graph, start, goal, heuristic):
    open_list = [(0, start, [])]
    closed_list = set()

    while open_list:
        # Extraer el nodo con menor costo total de la "open list"
        cost_total, current_node, path = heapq.heappop(open_list)

        if current_node == goal:
            return path + [current_node]

        if current_node in closed_list:
            continue

        # Marcar el nodo actual como visitado
        closed_list.add(current_node)

        # Expandir los vecinos del nodo actual
        for neighbor, cost in graph[current_node]:
            if neighbor not in closed_list:
                g_cost = cost_total + cost  # Costo real desde el inicio hasta el vecino
                h_cost = heuristic(neighbor, goal)  # Costo estimado desde el vecino hasta el objetivo
                f_cost = g_cost + h_cost  # Costo total estimado (g + h)

                # Agregar el vecino a la "open list" con su costo total como prioridad
                heapq.heappush(open_list, (f_cost, neighbor, path + [current_node]))

    # Si no se encontró un camino al objetivo, devolver None
    return None
```