Lab: Agentes (III)

Sistemas Inteligentes Distribuidos

Sergio Alvarez
Javier Vázquez

Objetivos de la sesión

- Ver cómo implementar conceptos de BDI con AgentSpeak

Bucle de control de agente (versión 4)

```
B := B0;
I := I0;
while true do
    get next percept ρ;
    B := brf(B, ρ);
    D := options(B, I);
    I := filter(B, D, I);
    π := plan(B, I);
    while not (empty(π)
            or succeeded(I, B)
            or impossible(I, B)) do
        α := hd(π);
        execute(α);
        π := tail(π);
        get next percept ρ;
        B := brf(B, ρ);
        if reconsider(I, B) then
            D := options(B, I);
            I := filter(B, D, I);
        end-if
        if not sound(π, I, B) then
            π := plan(B, I)
        end-if
    end-while
end-while
```

revisión de creencias: brf
actualización de deseos: options
deliberación: filter
medios – fines: plan
reconsideración de planes: reconsider/sound

AgentSpeak no implementa este bucle de control y es más flexible, pero permite implementar los diferentes pasos de manera lógica

Implementando BDI con AgentSpeak

Revisión de creencias

- añadir/borrar creencias
  - `+belief(x), -belief(x)`
- reglas de inferencia
  - `derived(x) :- belief(x)`

Actualización de deseos

- añadir nuevo objetivo a cumplir inmediatamente
  - `!g`
- añadir nuevo objetivo no prioritario
  - `!!g`
- puede ser en diferentes fases: como reacción a un evento, o como parte de un plan

Deliberación

- condiciones en el contexto de los planes que cumplen el objetivo
  - `+!g : can_run_g <- plan.`
  - `+!g: true <- !!g`

Medios – fines

- condiciones en el contexto de los planes
  - `+!g : condition_1 <- plan.`
  - `+!g : condition_2 <- plan.`

Reconsideración de planes

- estrategias de commitment: blind vs single-minded

Ejecutar AgentSpeak sin SPADE

- Descargad el fichero sesion3.zip y descomprimid
- Para ejecutar cada ejemplo:
  - Activad el entorno virtual
  - Ejecutad: python -m agentspeak <fichero>.asl
- Este método no carga el agente en la plataforma, por lo que:
  - No tenéis acceso a las acciones ni percepciones del entorno pyGOMAS
  - No existe el concepto de agente por lo que no hay comunicación ni acceso a los beliefs del agente desde Python
- Cada fichero presenta patrones que pueden servir (combinados) para construir un agente guiado por objetivos

00_belief_revision.asl

```
derived_belief_2 :- new_belief_2.

!add_new_percept.

+!add_new_percept <-
    +new_belief_1;
    +new_belief_2;
    !review_inferences.

+new_belief_1 <-
    .print("adding new belief due to new_belief_1");
    +derived_belief_1.

+!review_inferences : derived_belief_2 <-
    .print("adding new belief due to derived_belief_2");
    +derived_derived_belief_2.
```

- Se puede hacer revisión de creencias:
  - Añadiendo nuevas creencias
  - Derivando nuevas creencias a partir de eventos de creación/borrado de creencias
  - A partir de reglas de inferencia (pero sólo para objetivos de conocimiento o condiciones de contexto)

01_achievement_goal.asl

```
!g. // goal declaration
c1. // context (a fact)

+!g: g <- .print("goal achieved").
+!g: c1 <- !sg1; !g. // plan for g under context {c1}
+!g: c2 <- !sg2; !g. // plan for g under context {c2}
// ...
+!g: cn <- !sgn; !g. // plan for g under context {cn}

+!sg1 <- .print("executing subgoal1"); +g.
+!sg2 <- .print("executing subgoal2"); +g.
// ...
+!sgn <- .print("executing subgoaln"); +g.
```

- `!g` es un objetivo de logro que se puede cumplir de diferentes maneras
  - Cada manera de cumplirlo es condicional (`c1, …, cn`)
- Cada plan añade de nuevo el objetivo, para comprobar si se ha cumplido o no (g)
- Un plan en ejecución se considera intención en AgentSpeak

02_failed_goal.asl

```
!g. // goal declaration
    // but now there is no context!

+!g: g <- .print("goal achieved").
+!g: c1 <- !sg1; !g. // plan for g under context {c1}
+!g: c2 <- !sg2; !g. // plan for g under context {c2}
// ...
+!g: cn <- !sgn; !g. // plan for g under context {cn}
+!g <- .print("goal g has failed!").

+!sg1 <- .print("executing subgoal1"); +g.
+!sg2 <- .print("executing subgoal2"); +g.
// ...
+!sgn <- .print("executing subgoaln"); +g.
```

- `!g` es un objetivo de logro que se puede cumplir de diferentes maneras
  - Cada manera de cumplirlo es condicional (`c1, …, cn`)
- En este caso no tenemos contexto, por lo que ninguna condición de plan se cumple
- Necesitamos un evento `+!g` con condición true para gestionar el fallo

03_backtrack_wrong.asl

```
!g1.
!g2.

+!g1: g1 <- .print("goal g1 achieved").
+!g2: g2 <- .print("goal g2 achieved").

+!g1 <- .print("goal g1 has failed!"); !g1.
+!g2 <- .print("goal g2 has failed!"); !g2.
```

- `!g1` y `!g2` son objetivos de logro
- Al no poderse cumplir, tenemos que añadir de nuevo el objetivo
- Sin embargo, si añadimos con `!g1` estamos dando prioridad a este objetivo y no damos la oportunidad de cumplir con `!g2`
- Observad en el output cómo sólo se está probando `!g1`

04_backtrack_correct.asl

```
!g1.
!g2.

+!g1: g1 <- .print("goal g1 achieved").
+!g2: g2 <- .print("goal g2 achieved").

+!g1 <- .print("goal g1 has failed!"); !!g1.
+!g2 <- .print("goal g2 has failed!"); !!g2.
```

- Para permitir probar !g2 mientras `!g1` no se puede cumplir, podemos usar la forma `!!g1`, que encola el objetivo en lugar de ejecutarlo inmediatamente.

05_backtrack_tracking_wrong.asl

```
!g. // goal declaration
c1. // context (a fact)
!test_g_again.

+!g: solution(X) & g <- .print("goal achieved using
solution", X).
+!g: not solution(1) & c1 <- !sg1; +solution(1); !g.
// plan for g under context {c1}
+!g: not solution(2) & c2 <- !sg2; +solution(2); !g.
// plan for g under context {c2}
// ...
+!g: not solution(n) & cn <- !sgn; +solution(n); !g.
// plan for g under context {cn}

+!sg1 <- .print("executing subgoal1"); +g.
+!sg2 <- .print("executing subgoal2"); +g.
// ...
+!sgn <- .print("executing subgoaln"); +g.

+!test_g_again <- -c1; -g; +c2; !g.
```

- En este ejemplo queremos hacer seguimiento de qué opción ha permitido cumplir el objetivo
- Para eso definimos una creencia `solucion(X)` que toma un valor diferente dependiendo del plan
- El objetivo `!test_g_again` se usa para probar dos soluciones diferentes, al cambiar el contexto

05_backtrack_tracking_wrong.asl

```
!g. // goal declaration
c1. // context (a fact)
!test_g_again.

+!g: solution(X) & g <- .print("goal achieved using
solution", X).
+!g: not solution(1) & c1 <- !sg1; +solution(1); !g.
// plan for g under context {c1}
+!g: not solution(2) & c2 <- !sg2; +solution(2); !g.
// plan for g under context {c2}
// ...
+!g: not solution(n) & cn <- !sgn; +solution(n); !g.
// plan for g under context {cn}

+!sg1 <- .print("executing subgoal1"); +g.
+!sg2 <- .print("executing subgoal2"); +g.
// ...
+!sgn <- .print("executing subgoaln"); +g.

+!test_g_again <- -c1; -g; +c2; !g.
```

- En este ejemplo queremos hacer seguimiento de qué opción ha permitido cumplir el objetivo
- Para eso definimos una creencia `solucion(X)` que toma un valor diferente dependiendo del plan
- El objetivo `!test_g_again` se usa para probar dos soluciones diferentes, al cambiar el contexto
- Sin embargo, algo no está yendo bien y nos da la misma solución, **¿por qué?**
  - **Solución en `06_backtrack_tracking_correct.asl`**

07_contingency_plan.asl

```
!g1. // initial goal

+!g3: g <- .print("g3 achieved!").

+!g1: true <- !g2; .print("end g1").
+!g2: true <- !g3; .print("end g2").
+!g3: not fail <- !g4; .print("end g3"); !!g3.
+!g4: true <- !g5; .print("end g4").
+!g5: true <- .print("end g5 without reaching goal");
+fail.

+!g3: fail <- .print("handling g3 failure"); +g; !!g3.
```

- `!g1` es un objetivo cuyo cumplimiento se puede deconstruir en los subobjetivos `!g1 -> !g2 -> !g3 -> !g4 -> !g5`
- Sin embargo, `!g5` acaba sin cumplir el objetivo
- En este código se propone una manera de capturar el fallo en un subobjetivo intermedio, `!g3`
  - Esto permitiría deliberar o replanificar

08_blind_commitment.asl

```
g :- a1 | a2.
fail :- a3.

!g.
c3.

+!g: g <- .print("goal g has succeeded").

+!g : c1 & not fail <- +a1; !g.
+!g : c2 & not fail <- +a2; !g.
+!g : c3 & not fail <- +a3; !g.

+!g: fail <- -a1; -a2; -a3; -fail; !!g.
// in case of some failure
+!g: true <- !g. // keep trying
```

- `!g` se puede cumplir con uno de tres planes condicionales
- Con un predicado fail podemos controlar si ha habido un error en la ejecución de un plan
- Blind commitment: si el plan no se ha cumplido, volver a declarar `!g`
- ¿Qué ocurre cuando ejecutáis? **¿Por qué?**
  - **En 09_blind_commitment_success.asl tenéis una alternativa que soluciona este problema**

10_single_minded_commitment.asl

```
g :- a1 | a2.
fail :- a3.

!g.
c3.

+!g: g <- .print("goal g has succeeded").

+!g : c1 & not fail <- +a1; !g.
+!g : c2 & not fail <- +a2; !g.
+!g : c3 & not fail <- +a3; !g.

+!g: fail <- -a1; -a2; -a3; -fail; .print("goal g has
failed"); !g2. // goal has failed
+!g: true <- !g. // keep trying

+!g2 <- .print("goal g2 has succeeded").
```

- En este caso tenemos compromiso de determinación (single-minded commitment): mantenemos la intención mientras el objetivo no falle o tenga sentido replanificar

Cómo diseñar los agentes

- Os recomendamos seguir los siguientes pasos antes de empezar a implementar:
  - Para cada rol, definid qué entendéis como “comportamiento racional”, por ejemplo:
    - ¿En qué orden se debería actuar para capturar la bandera? ¿Primero cogerla, o asegurarse de que no hay enemigos cerca?
    - ¿En qué momento hay que producir un pack médico/de munición?
    - ¿Cómo de relevante es la vida que le queda al agente? ¿A partir de qué cifra hay que intentar huir de una batalla?
  - Identificad qué creencias, ya sea proporcionadas por el entorno como inferidas, son útiles para operacionalizar este comportamiento racional
  - Enumerad objetivos de alto nivel que representen este comportamiento, sin entrar todavía en planes concretos. Divididlos en subobjetivos si es necesario y definid una jerarquía
  - Teniendo en cuenta las propiedades del entorno y las acciones disponibles, diseñad los planes que podrían cumplir cada objetivo y en qué condiciones se escogería cada plan
- Para implementar, podéis inspiraros en, o usar código de los ejemplos en sesion1_2.zip, pero tened en cuenta que no son agentes muy eficientes y puede ser menos complicado hacer el código desde cero