多智能体系统 人工智能学院

第二次编程作业

Instructor: 章宗长 *Name:* 方盛俊, *StudentId:* 201300035

实验报告

2.1

给机器人 Agent 加入目标来源限制如下:

```
@h1
+!has(owner,beer)[source(S)]
   :  (S == self | S == owner) & available(beer,fridge)
      & not too_much(beer)
   <- !at(robot,fridge);
      open(fridge);
      get(beer);
      close(fridge);
      !at(robot,owner);
      hand_in(beer);
      ?has(owner,beer);
      // remember that another beer has been consumed
      .date(YY,MM,DD); .time(HH,NN,SS);
      +consumed(YY,MM,DD,HH,NN,SS,beer).

@h2
+!has(owner,beer)[source(S)]
   :  (S == self | S == owner) & not available(beer,fridge)
   <- .send(supermarket, achieve, order(beer,5));
      !at(robot,fridge). // go to fridge and wait there.

@h3
+!has(owner,beer)[source(S)]
   :  (S == self | S == owner) & too_much(beer) & limit(beer,L)
   <- .concat("The Department of Health does not allow me ",
              "to give you more than ", L,
              " beers a day! I am very sorry about that!",M);
      .send(owner,tell,msg(M)).
```

可以看出, 和原来的家政机器人的代码的区别在于, 我们通过 Source 标注对来源进行标注, 即 `+!has(owner,beer)[source(S)]` 获取到来源 `S` 后, 再通过上下文条件 `(S == self | S == owner)` 将来源限制为 `self` 或 `owner`.

通过标注来源为 `self` 或 `owner`, 即使我们加入一个 `attacker` agent, 也无法对家政机器人发起目标.

2.2

修改超市 Agent 代码如下:

```
last_order_id(1). // initial belief
// 库存为 100
stock(beer, 8). // initial belief
// stock(beer, 100). // initial belief

// plan to achieve the goal "order" for agent Ag
+!order(Product,Qtd)[source(Ag)] : stock(beer, X) & X >= Qtd
  <- ?last_order_id(N);
     OrderId = N + 1;
     -+last_order_id(OrderId);
     deliver(Product,Qtd);
     -stock(beer, X);
     +stock(beer, X - Qtd);
     .print("Stock of ", Product, " is ", X - Qtd);
     .send(Ag, tell, delivered(Product,Qtd,OrderId)).

+!order(Product,Qtd)[source(Ag)] : stock(beer, X) & X < Qtd
   <- .concat("There is no enough ", Product, ", only ", X,
              " but need ", Qtd, M);
      .print(M);
      .send(Ag,tell,msg(M)).
```

其中当初始库存为 `stock(beer, 8)` 时, 超市 Agent 的关键输出如下 (完整输出位于 `2-2/output_8.txt`):

```
[supermarket] doing: deliver(beer,5)
[supermarket] Stock of beer is 3
[supermarket] There is no enough beer, only 3 but need 5
```

其中当初始库存为 `stock(beer, 100)` 时, 超市 Agent 的关键输出为 (完整输出位于 `2-2/output_100.txt`):

```
[supermarket] doing: deliver(beer,5)
[supermarket] Stock of beer is 95
[supermarket] doing: deliver(beer,5)
[supermarket] Stock of beer is 90
[owner] Message from robot: The Department of Health does not allow me
        to give you more than 10 beers a day! I am very sorry about that!
```

2.3

(1)

将原先的两个规划的实现顺序调换, 不会对运行结果有影响, 两种方式最后的输出结果都是一致的, 并且 robot 均能正常移动.

原因是因为这两个规划的上下文 `at(robot,P)` 和 `not at(robot,P)` 是对立互斥的, 两条规划中必然有且仅有一条规划会被执行, 即使互换顺序也不会相互影响.

(2)

如果仅仅是改为右图的话, 对运行结果不会有影响. 因为程序会有限执行 `m1` 规划, 如果不满足 m1 规划的上下文, 则会去执行 m2 规划, 这与 (1) 中的必然有且仅有一条规划会被执行的做法是一致的.

但是如果我们再将两个规划的顺序调换, 运行结果就不正确了, 由于优先判断 m2 规划, 且 m2 规划的上下文为 `true`, 总是会被执行, 因此 robot 即使是达到目标位置后, 也总是处于 `move_towards(fridge)` 的动作中, 无法判断是否要执行 m1 规划, 也就无法认定自己实现了 `at(robot, fridge)` 这个目标, 因此不会停止.

2.4

我们给发起者 `c.asl` 加上初始目标 `!randomlyAbort(1, 0.5).` 让其以 0.5 的概率将任务取消, 取消方式是使用 `untell` 撤销 `cfp` 信念, 具体的代码如下:

```
+!randomlyAbort(CNPId, Prob)
   <- .wait(3000); // wait 3 seconds
      .random(R);
      if (R < Prob) {
        !abort(CNPId);
      } else {
        .print("CNP ",CNPId," continue.");
      }.

+!abort(CNPId)
   <- .findall(Name,introduction(participant,Name),LP);
      .print("Sending abort to ",LP);
      .send(LP,untell,cfp(Id,_));
      -+cnp_state(Id,aborted).
```

并为参与者 `p.asl` 设置好应答:

```
// answer to Abort Call For Proposal
@c2 -cfp(CNPId,Task)[source(A)]
   :  plays(initiator,A)
   <- .print("CNP ",CNPId," for ",Task," aborted.").
      -proposal(CNPId,_,_). // clear memory
```

当任务被取消时, 具体输出如下:

```
[c] Waiting participants...
[c] Sending CFP to [pn,p3,p1,pr,p2]
[c] Sending abort to [pn,p3,p1,pr,p2]
[p1] CNP 1 for fix(computer) aborted.
[p2] CNP 1 for fix(computer) aborted.
[p3] CNP 1 for fix(computer) aborted.
```

当任务继续执行时, 具体输出如下:

```
[c] Waiting participants...
[c] Sending CFP to [p2,p1,pr,p3,pn]
[c] CNP 1 continue.
[c] Offers are [offer(102.63511690002497,p1),offer(107.78038433412198,p3),
    offer(105.70901934736834,p2)]
[c] Winner is p1 with 102.63511690002497
[p3] I lost CNP 1.
[p2] I lost CNP 1.
[p1] My proposal '102.63511690002497' won CNP 1 for fix(computer)!
```

2.5

为了让参与者 `p.asl` 能够撤销自己之前提出的报价, 我们以 0.5 的概率随机地触发 `+cancel(CNPId)`:

```
// randomly cancel a CNP
!randomlyCancel(1, 0.5, 3000).
// !randomlyCancel(1, 0.5, 5000).

/* Plans */

+!randomlyCancel(CNPId, Prob, Delay)
   <- .wait(Delay); // wait 3 seconds
      .random(R);
      if (R < Prob) {
        +cancel(CNPId);
      } else {
        .print("I don't want to cancel ",CNPId,".");
      }.
```

在参与者 `p.asl` 触发了 `+cancel(CNPId)` 事件后, 会向发起者提出撤销申请, 并通过 `+cancel_succeeded(CNPId)` 和 `cancel_failed(CNPId)` 监听是否撤销成功, 成功则将自身的 `proposal(CNPId,_,_)` 清除.

```
// handle cancel
@r3 +cancel(CNPId)
   :  plays(initiator,A)
   <- .print("I want to cancel ",CNPId);
      .send(A,tell,cancel(CNPId)).

@r4 +cancel_succeeded(CNPId)
   <- .print("I successfully canceled ",CNPId,".");
      -proposal(CNPId,_,_). // clear memory

@r5 +cancel_failed(CNPId)
   <- .print("I failed to cancel ",CNPId,").
```

在发起者 `p.asl` 接收到 `+cancel(CNPId)` 事件后, 会通过上下文中的 `cnp_state(CNPId,propose)` 判断是否处于 propose 状态, 如果成功则删除 `propose(CNPId,_)[source(Ag)]`, 否则只能告知 `Ag` 撤销报价失败了.

```
// receive cancels
@r3 +cancel(CNPId)[source(Ag)]
   :  cnp_state(CNPId,propose)
   <- -propose(CNPId,_)[source(Ag)];
      .send(Ag,tell,cancel_succeeded(CNPId)).

@r4 +cancel(CNPId)[source(Ag)]
   :  not cnp_state(CNPId,propose)
   <- .send(Ag,tell,cancel_failed(CNPId)).
```

当我们设置 `Delay = 3000` 时, 一种输出如下:

```
[c] Waiting participants...
[c] Sending CFP to [p2,pn,p3,p1,pr]
[p2] I don't want to cancel 1.
[c] CNP 1 continue.
[p1] I want to cancel 1
[p3] I don't want to cancel 1.
[p1] I successfully canceled 1.
[c] Offers are [offer(100.96298725170662,p3),offer(105.42136746362259,p2)]
[c] Winner is p3 with 100.96298725170662
[p3] My proposal '100.96298725170662' won CNP 1 for fix(computer)!
[p2] I lost CNP 1.
```

可以看出, 由于 3 秒的延迟在时间限制内, 发起者还处于 propose 状态, 因此 p1 成功地取消了自己之前提出的报价, 可以看到 Offers 中只包含了 p2 和 p3 的报价.

当我们设置 `Delay = 7000` 时, 一种输出如下:

```
[c] Waiting participants...
[c] Sending CFP to [pn,p3,p1,pr,p2]
[c] CNP 1 continue.
[c] Offers are [offer(104.83744062931348,p1),offer(107.48395053403816,p3),
    offer(109.14706668300498,p2)]
[c] Winner is p1 with 104.83744062931348
[p1] My proposal '104.83744062931348' won CNP 1 for fix(computer)!
[p2] I lost CNP 1.
[p3] I lost CNP 1.
[p3] I don't want to cancel 1.
[p1] I want to cancel 1
[p2] I want to cancel 1
[p1] I failed to cancel 1.
[p2] I failed to cancel 1.
```

可以看出, 由于 7 秒的延迟不在时间限制内, 发起者已经处于 contract 或 finished 状态了, 因此 p1 和 p2 没有能够取消自己之前提出的报价.