UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

INE5430 - Inteligência Artificial

Profª Jerusa Marchi

Relatório Sistemas Multiagente

Bruna Carpes de Freitas (18200556)
Larissa Gremelmaier Rosa (20100531)
Maren Freland Arn (941936)
Raquel Cristina Schaly Behrens (20100544)

28 de maio de 2023

Item 1

Faça os robôs mineradores colaborarem, ou seja, quando um robô encontrar o recurso desejado, faça com que ele notifique os demais robôs sobre a posição do recurso, assim todos os robôs devem coletar o recurso no mesmo local. Porém, um robô que já está coletando recursos em outro local, não deveria abandonar a sua meta atual.

Para resolver o primeiro item, adicionamos mais situações para *'check_for_resources'*, onde, se o recurso necessário for R, e ter chegado em alguma posição em que R existe, esse recurso é notificado para os outros agentes, por meio do *broadcast*, e o agente que encontrou o recurso passa a captar o recurso para o *boss*:

```
32 //If wants a resource R, and found this R, start taking it and let the others know
33 +!check_for_resources
34     : resource_needed(R) & found(R) & my_pos(X, Y)
35     <- !stop_checking;
36        .broadcast(tell, resource(R, X, Y));
37        !take(R, boss);
38        !continue_mine.
```

Os outros agentes, então, recebem esse broadcast:

```
74 //Being notified of a resource, and is neither after another resource nor is in the middle of something, get resource
75 +resource(R, X, Y)
76     : not my_res(_, _, _) & not pos(back, _, _) & checking_cells
77     <- !get_resource(R, X, Y).
78
79 //If if not after another resource, go get the resource and record that is after a resource
80 +!get_resource(R, X, Y)
81     : not my_res(_, _, _)
82     <- +my_res(R, X, Y);
83        // Agent stop checking
84        !stop_checking;
85        // Agent goes to resource
86        !go(R, X, Y).
87
```

Assim, ao receber um recurso, que está em X e Y, caso o agente não esteja coletando um recurso (ou seja, não tenha um *my_res(_,_,_))*, não esteja fazendo algo que depois exija que volte para sua posição original (ou seja, não tenha um *pos(back, _,_)*, e não esteja coletando algum recurso (ou seja, tenha *checking_cells)*, o agente parte para coletar esse recurso. Assim, garantimos que um robô que esteja coletando recursos em outro local, não abandone sua meta atual.

Em !*get_resource*, chamamos *!go(R, X, Y)*. Criamos três condições para esse *go:*

```
100 //If has been told to go after a resource, and is in this position, and found this resource, take this to the boss and come back
101 +!go(R, X, Y)
102     : my_pos(X, Y) & found(R)
103     <- !take(R, boss);
104        !go(R, X, Y).
105
106 //If has been told to go after a resource, and is in this position, but did not find the resource, let the others know that the re
107 +!go(R, X, Y)
108     : my_pos(X, Y) & not found(R)
109     <- .broadcast(untell, resource(R, X, Y));
110        .broadcast(untell, my_res(R, X, Y));
111        .broadcast(untell, resource_history(R, X, Y));
112        -resource_history(R, X, Y);
113        -resource(R, X, Y);
114        -my_res(R, X, Y);
115        !continue_mine.
116
117 //If has been told to go after a resource, and is not in the position yet, move towards this position and continue going
118 +!go(R, X, Y) : true
119     <- move_towards(X, Y);
120        !go(R, X, Y).
```

O primeiro é: Se o agente foi encarregado de ir atrás de um recurso (ou seja, possui *go(R, X, Y)* como evento), e já está na posição passada, e encontrou esse recurso, então levará o recurso para o *boss* e voltará para o local novamente.

O segundo é: Se o agente foi encarregado de ir atrás de um recurso (ou seja, possui *go(R, X, Y)* como evento), e já está na posição passada, mas não encontrou o recurso, quer dizer que o recurso acabou. Portando, avisa os outros agentes, com um *broadcast*, para que deletem de suas crenças que existe recurso naquele local, que existe recurso futuro naquele local, e que ainda existe o recurso que algum agente já está pegando naquele local. E, após isso, continua a procurar recursos.

O terceiro é: Se o agente foi encarregado de ir atrás de um recurso (ou seja, possui *go(R, X, Y)* como evento), e não entrou na primeira e nem na segunda condição, move-se em direção à posição X, Y, e encarrega-se de ir até o recurso novamente.

Além disso, modificamos *-resource* para deletar juntamente consigo se está pegando o recurso passado (*my_res)* e se possui esse recurso em seu histórico de futuros recursos a serem pegados (*resource_history)*, e continua a procurar minérios. Isso foi feito para garantir que os resquícios de um recurso que não existe mais, vão ser deletados.

```
88 //Being notified that can exclude record of a resource, if
89 -resource(R, X, Y)
90     : resource(R, X, Y) | my_res(R, X, Y)
91     <- .drop_desire(go(R, X, Y));
92        .broadcast(untell, resource(R, X, Y));
93        .broadcast(untell, my_res(R, X, Y));
94        .broadcast(untell, resource_history(R, X, Y));
95        -resource_history(R, X, Y);
96        -resource(R, X, Y);
97        -my_res(R, X, Y);
98        !continue_mine.
99
```

Item 2

Ainda no contexto de colaboração, faça o robô que percebeu o esgotamento do recurso em determinado local avisar aos demais sobre o esgotamento desse recurso.

Para garantir isso, criamos outra condição em *!check_for_resources*: se o agente está procurando o recurso R (ou seja, já tinha o encontrado nessa posição, e criou a crença *my_res(R, X, Y))*, está na posição que está procurando, porém não encontrou o recurso, quer dizer que o recurso daquela posição acabou. Para isso, notifica os outros agentes que o recurso acabou naquele local, deleta todas as suas crenças acerca desse recurso, e passa a continuar procurando recursos.
A segunda condição: Faz as mesmas coisas que a primeira condição, mas se não encontrou um recurso do qual possuía registro, mesmo não estando atrás desse recurso, deleta-o e notifica todos os outros agentes que aquele recurso não existe mais naquela posição.

```
40 //If was looking for a resource R, is in it's place, but did not find resource
41 +!check_for_resources
42     : not found(R) & my_pos(X, Y) & my_res(R, X, Y)
43     <- .broadcast(untell, resource(R, X, Y));
44        .broadcast(untell, my_res(R, X, Y));
45        .broadcast(untell, resource_history(R, X, Y));
46        -resource_history(R, X, Y);
47        -my_res(R, X, Y);
48        -resource(R, X, Y);
49        move_to(next_cell);
50        !continue_mine.
```

Item 3

Permitir aos robôs guardarem informações sobre os recursos localizados, mesmo que ainda não estejam sendo procurados. Assim, ao precisar de um novo recurso, os robôs poderão (talvez) já ter informações de onde o recurso se encontra. Toda vez que um recurso (mesmo não procurado) for encontrado, também faça com que os robôs troquem estas informações.

Para esse ítem, criamos outras condições em *!check_for_resources*: se o agente

precisa de um recurso R, mas encontrou outro recurso S, e S é um recurso futuro (ou seja, seu índice é maior que o recurso atual), então guarda uma crença do histórico de recursos encontrados (*+resource_history(S, X, Y)*), e manda essa crença para os outros agentes por meio de um *broadcast*.

```
9  //If wants resource R, but found S, and S is a future resource, then tell all a
10 +!check_for_resources
11     : resource_needed(R) & found(S) & R < S & my_pos(X, Y)
12     <- +resource_history(S, X, Y);
13        .broadcast(tell, resource_history(S, X, Y));
14        move_to(next_cell).
```

Assim, caso o agente agora esteja precisando de um recurso futuro, e há uma crença desse recurso *resource_history(R, X, Y),* então sinaliza para os outros buscarem esse recurso por meio de um *broadcast*, sinaliza para os outros tirarem esse recurso de seu histórico por meio de um *broadcast*, tira esse recurso do seu histórico, e vai atrás do recurso por *!get_resource*.

```
24 //If wants resource R, and has a record of history, go to get resource and tell others
25 +!check_for_resources
26     : resource_needed(R) & resource_history(R, X, Y)
27     <- .broadcast(tell, resource(R, X, Y));
28        .broadcast(untell, resource_history(R, X, Y));
29        -resource_history(R, X, Y);
30        !get_resource(R, X, Y).
```

Além disso, se algum recurso foi mandado para um agente, mas ele não pode ir atrás desse recurso naquele momento, criou-se outra condição para *!check_for_resources*, onde caso um agente precise de um recurso, e esse recurso ainda está como um recurso enviado para o mesmo, o agente vai atrás deste recurso:

```
16 //If wants resource R, and has a record of resource R in certain location, go get resource and tell others
17 +!check_for_resources
18     : resource_needed(R) & resource(R, X, Y)
19     <- !stop_checking;
20        .broadcast(tell, resource(R, X, Y));
21        !get_resource(R, X, Y);
22        !continue_mine.
```

Item 4

Note que, após colaborar na coleta de um recurso, o agente volta a navegar "aleatoriamente"no ambiente. Faça com que ele retorne para a posição em que estava antes de ser chamado a ajudar na coleta.

Garantimos que o agente volte para sua posição original, após ser chamado para minerar em outro local, guardando essa posição como *pos(back,X,Y),* em *!stop_checking*, que é chamado toda vez que recebe um broadcast para pegar um recurso em outro local*:*

```
120 +!stop_checking : not pos(back, _, _)
121     <- ?my_pos(X, Y);
122        +pos(back, X, Y);
123        -checking_cells.
124
```

E, ao terminar de coletar o recurso em outro local, o agente volta para essa posição *pos(back, X, Y),* por meio de *!go(back)*, em *!continue_mine*, que é invocado toda vez que o agente termina de coletar o recurso desse outro local.

```
134 +!continue_mine : pos(back, X, Y)
135     <- !go(back);
136        -pos(back,X,Y);
137        +checking_cells;
138        !check_for_resources.
```