Projeto Rescue Simulator

Autores

Luan Carlos Klein
Cesar Augusto Tacla

UTFPR, Câmpus Curitiba, grupo PET ENGENHARIA DE COMPUTAÇÃO

RESCUE SIMULATOR

Permite construir um ambiente na forma de um labirinto onde há um agente. A arquitetura do agente é baseada no modelo BDI (*Beliefs, Desires and Intentions*).

O agente possui um ciclo de raciocínio e a cada iteração recebe percepções do ambiente por meio dos seus sensores, processa estas percepções para verificar o estado atual do ambiente e construir uma representação do estado atual do ambiente. Com esta representação atualizada e com suas crenças (fornecidas pelo programador ou inferidas), delibera sobre a próxima ação que o levará mais próximo do estado objetivo. Tendo escolhido a ação, o agente atua no ambiente modificando o estado deste último. A partir daí o ciclo se reinicia.

CLASSES E MÉTODOS PRINCIPAIS

Main

Faz a instanciação das demais classes. Contém o *loop* de controle principal (**o ciclo de raciocínio do agente**). A cada iteração, invoca o método de deliberação do agente até que o agente decida pelo término.

Model

É o ambiente onde o agente está situado. No caso do RobôFun, o ambiente é um labirinto com paredes e vítimas. Quando o agente executa uma ação, normalmente uma mudança de estado se produz no *model*. Por exemplo, se o agente se desloca para uma nova posição, isto deve refletir no *model* que atualizará a informação da posição do agente. Em teoria, o agente somente acessa os atributos desta classe por meio dos seus sensores e age por meio dos seus atuadores. É interessante manter esta independência para que seja fiel a situação do agente não ter acesso a todas as variáveis do ambiente sem que esteja equipado para isto (ambiente parcialmente observável).

<Agent>

Exemplos disponíveis no projeto: agentRnd.py

O agente atua no ambiente e o modifica, portanto, necessita de uma referência ao mesmo: **`self.model`**

No agente, criamos uma instância do problema a ser solucionado, o que denominamos de formulação do problema – e.g. um problema de busca de caminho em um mapa (ver a classe <Plan>

Exemplos disponíveis no projeto: RandomPlan.py

Um plano é uma sequência de ações. Se o agente está em um ambiente estático e determinístico e tem o tempo que necessitar para planejar então pode se utilizar de uma

estratégia de busca cega ou informada e calcular todas as ações necessárias para sair do estado inicial e atingir o estado objetivo.

Se adicionarmos uma limitação de tempo, ou seja, o agente tem que intercalar deliberação e ação, então estamos no caso dos algoritmos on-line embora o ambiente possa ainda ser estático e determinístico. Neste caso, o plano será uma ação, ou seja, o agente computa apenas a próxima ação.

Problem).

O método de deliberação é o procedimento no qual o agente decide qual é a próxima ação que vai executar no ambiente:
`deliberate(self)`

Na versão atual, um agente pode perseguir vários objetivos de forma sequencial. Para cada objetivo, possui um plano possivelmente construído por algoritmos diferentes ou porque tem uma série de objetivos a perseguir.

Por exemplo, um agente pode ter:

1) plano para realizar movimentos aleatórios até gastar um porcentual da sua bateria
2) plano construído pelo A* para ir de um estado inicial até um estado objetivo rapidamente (caminho de menor custo) e,
3) finalmente, um plano para retornar à base utilizando LRTA*.

Os planos a serem executados por um agente ficam armazenados em uma biblioteca de planos chamada **`libPlan`** e são executados na ordem em que são inseridos nesta lista.

A metáfora dos planos é utilizada também para estratégias de busca on-line. Neste caso, o plano retornará uma ação por vez quando chamado a cada ciclo de raciocínio do agente.

Se a estratégia de busca for off-line e o ambiente for estático e determinístico, o plano é calculado somente uma vez. A cada deliberação, basta o agente pegar a próxima ação a ser executada até o seu término quando terá atingido o objetivo.

<Plan>

Exemplos disponíveis no projeto: RandomPlan.py

Um plano é uma sequência de ações. Se o agente está em um ambiente estático e determinístico e tem o tempo que necessitar para planejar então pode se utilizar de uma estratégia de busca cega ou informada e calcular todas as ações necessárias para sair do estado inicial e atingir o estado objetivo.

Se adicionarmos uma limitação de tempo, ou seja, o agente tem que intercalar deliberação e ação, então estamos no caso dos algoritmos on-line embora o ambiente possa ainda ser estático e determinístico. Neste caso, o plano será uma ação, ou seja, o agente computa apenas a próxima ação.

Problem

Contém a modelagem ou a formulação do problema adaptado ao problema de trajetórias em labirintos. Portanto, um estado corresponde a posição atual do agente dada por coordenadas (row, col). A classe State define métodos para instanciação e atribuição de valores a um

estado. Na classe Problem, chamamos os métodos para definir o estado inicial, objetivo e um teste de objetivo.

Estado inicial:
`defInitialState(self, row, col)`

Estado objetivo:
`defGoalState(self, row, col)`

Teste de objetivo: permite adicionar outros critérios além da posição do agente, por exemplo, estar em uma posição com algum objeto específico.
`goalTest(self, currentState)`

EXEMPLO INICIAL

Um labirinto é utilizado para representar um cenário pós-desastre: as paredes representam barreiras intransponíveis (escombros, lama) e as vítimas estão dispersas no ambiente sem possibilidade de se locomoverem. Um agente anda aleatoriamente no labirinto e, ao encontrar uma vítima, faz a leitura dos sinais vitais. O ciclo de raciocínio do agente está ilustrado no diagrama de sequência abaixo para uma estratégia on-line.

Main	Agent	Plan	Model
Instancia model			
			Cria representação do ambiente conforme arquivos config.txt, ambiente.txt e vitimas.txt
Instancia agente			
	Instancia Problem e Plan		
agent.deliberate()			
	Verifica se a última ação produziu o estado esperado e atualiza a posição atual: Self.positionSensor()		
			Atributo model.agentPos
	Adiciona o custo da última ação ao custo total de execução do plano		
	Verifica se atingiu o custo máximo		
	Verifica se há vítima na posição atual: self.victimPresenceSensor()		
			isThereVictim() retorna id da vítima
	Lê sinais vitais pelo id: victimVitalSignalSensor(id)		
			getVictimVitalSignals(id) retorna lista de sinais
	Processa os sinais vitais		
	Esolhe próxima ação: self.plan.chooseAction()		
		chooseAction(): escolhe posição aleatória dentro do labirinto sem que tenha parede, retorna ação e estado esperado	
	Executa a ação: self.executeGo(<direção>)		
			go(action)
agent.deliberate()			
	…	…	…