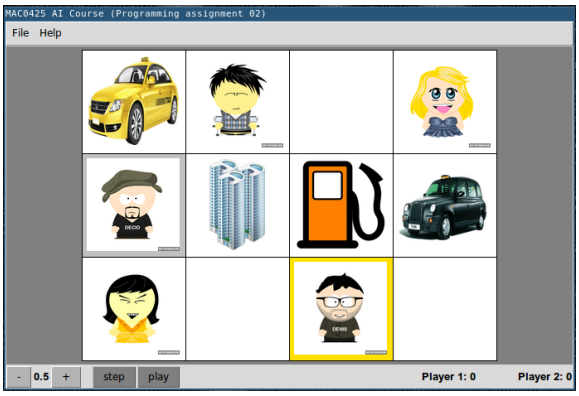
Segundo Exercício-Programa: USPber

Entrega: 01/05/2019

Motivação

Sua tarefa nesse EP é desenhar agentes inteligentes para um jogo baseado em turnos no qual dois jogadores competem pela coleta de passageiros. Você estará sempre encarregado do jogador 1, o primeiro a jogar, e deverá testar diversas estratégias. O jogo é jogado em um tabuleiro (*gridworld*) retangular, cujas células podem conter passageiros de diferentes tipos, postos de abastecimento ou obstáculos como casas etc. A figura abaixo exibe a interface gráfica do jogo (à direita) e uma representação matricial interna. Note que a matriz segue o sistema de coordenadas de imagens, ou seja, a coordenada $(0, 0)$ corresponde a célula do canto superior esquerdo.

```
Example:
(0,0)                 (0,4)
-------------------------          List with some elements:
| 1 | 0 | 0 | 0 | 3 |              (0,0): Car01
-------------------------          (4,4): Car02
| 0 | 0 | 3 | 0 | 0 |              (0,4),(1,2): Students
-------------------------          (2,2): Gas station
| 0 | 5 | 4 | 5 | 0 |              (2,1),(2,3): buildings
-------------------------          (3,1): Professor
| 0 | 6 | 0 | 0 | 0 |              (4,3): Monitor
-------------------------          The rest is empty
| 3 | 0 | 0 | 7 | 2 |
-------------------------
(4,0)                 (4,4)
```



Descrição do jogo

Os elementos do jogo são representados matricialmente por números inteiros entre 0 e 9 (sendo 8 e 9 reservados para a situação que um jogador está parado em um posto de reabastecimento). A descrição dos códigos é dada abaixo:

Code:6 Code:7 Code:7 Code:7 Code:3 Code:1 Code:5 Code:4

Code:3 Code:3 Code:3 Code:3 Code:0 Code:2 Code:5

A cada jogada, um jogador executa uma ação de mover para uma célula vizinha (UP, RIGHT, DOWN, LEFT), de reabastecer (REFILL) ou de permanecer parado (STOP). A ação

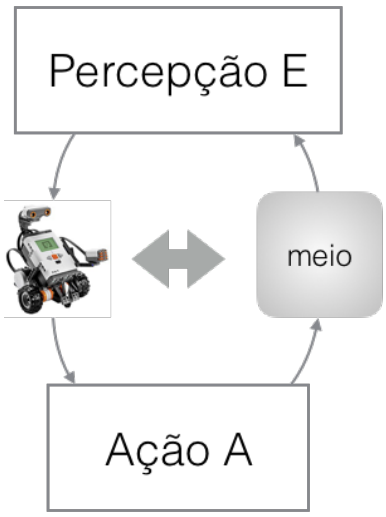
de reabastecer só é aplicável em uma célula contendo um posto de reabastecimento, aumenta o combustível em `util.DEFAULT_REFILL` unidades (até o limite máximo), e é efetivada no turno seguinte. As ações de movimentação só são aplicáveis se o veículo possuir combustível, a célula correspondente existe, e não possui obstáculo. A cada movimento o tanque tem sua capacidade decrementada de uma unidade. Ao se mover para uma célula com um passageiro, este é coletado e os pontos equivalentes são atribuídos ao jogador correspondente. O jogo acaba se: Todas as pessoas foram coletadas; ou ambos os veículos estão sem combustível e não podem abastecer; ou o número de turnos do jogo foi superior a `util.MAX_TURNS`.

Antes de começar

Você deverá implementar sua solução no esqueleto de código fornecido no arquivo `ep2.py`. Você pode criar novas funções, mas não deve modificar o protótipo das funções existentes. Além disso, você deve respeitar os casos de herança de classes abstratas explicitadas no esqueleto.

Parte 0: Agentes e Ambiente

Um agente é uma entidade autônoma capaz de perceber o meio (*environment*) e modificá-lo através de uma ação. Essa interação entre agente e meio é comumente modelada na arquitetura do agente como um ciclo percepção-ação como mostra a figura abaixo.



Neste EP, adotaremos o modelo de agente com ciclo percepção-ação, que é implementado no arquivo `game.py`. Nesse arquivo também é implementado o controlador do jogo, que instancia os agentes com as informações necessárias, fornece a percepção do ambiente a cada turno (contendo a matriz de representação do tabuleiro e o nível de combustível no seu tanque) e recebe e executa uma ação do agente.

Testando a interface

Sua primeira tarefa é verificar o funcionamento do jogo com agente pré programados e se familiarizar com a interface, executando o comando:

```
python3 game.py
```

Você deve observar uma execução do jogo na interface gráfica. Na parte inferior da interface existem botões + e − que controlam a velocidade da simulação, um botão `Step` que avança um

turno, e um botão `Play` que inicia a simulação até o fim do jogo. Ao final do jogo, é apresentado um sumário com informações gerais sobre o jogo.

Seu primeiro agente

Os agentes padrões que foram usados na simulação anterior escolhem uma ação de forma aleatória, mesmo que ela não seja aplicável, levando a situações indesejadas. Você deve editar o arquivo `ep2.py`, modificando o método `get_action` para que o `RandomAgent` não escolha mais ações inválidas, ou seja, para que ele escolha aleatoriamente uma ação apenas dentre o conjunto de ações aplicáveis naquela configuração de jogo. Para isso, você deve analisar a matriz de representação do jogo passada ao agente ao começo do turno.

Ao terminar sua implementação, execute novamente o arquivo `game.py`.

Parte 1: Agentes planejadores

Um agente aleatório é por certo uma estratégia ruim para seguir. Um agente mais efetivo consiste em escolher uma ação de forma a se aproximar de alguma meta útil para a obtenção de pontos. Uma possibilidade é estabelecer como meta coletar algum passageiro, e escolher uma ação através de um problema de busca. Esse tipo de estratégia, na qual o agente formula uma meta e obtém uma ação planejando para atingir tal meta, é conhecida como agente planejador. Como o meio pode não corresponder ao modelo (problema de busca) do agente, o agente precisa *replanejar* suas ações a cada passo, ou seja, a cada turno o agente planejador formula um problema de busca baseado na configuração atual do jogo, encontra um plano ótimo e executa a primeira ação desse plano.

O agente `GetClosestPersonOrRefillAgent` contém uma implementação de um agente planejador que busca coletar o passageiro mais próximo, formulando o problema como uma busca determinística em espaço de estados resolvida usando uma busca A* em grafo. Para ver esse agente em ação, abra o arquivo `game.py` e localize no final do arquivo as seguintes linhas:

```
Player1Agent = ep2.RandomAgent
Player2Agent = ep2.RandomAgent
```

Essas linhas são responsáveis para informar ao jogo qual tipo de agente cada jogador está usando. Modifique essas linhas para:

```
Player1Agent = ep2.GetClosestPersonOrRefillAgent
Player2Agent = ep2.DoNothingAgent
```

Dessa forma, nosso agente jogará contra um oponente que permanece imóvel, e portanto o jogo satisfaz a hipótese de determinismo assumida pelo agente (essa é uma situação apenas para teste). Para ver o agente em ação, execute:

```
python3 game.py
```

Você verá que embora o jogador oponente permaneça imóvel, o jogo termina sem que o jogador 1 colete todos os passageiros. Isso ocorre porque o jogo está configurado para acabar no início do 11^o turno, o que não é o suficiente para coletar todos os passageiros. Para alterar isso, abra o arquivo `util.py` e altere o valor da constante global `MAX_TURNS` para 20.

Ao executar novamente o jogo, você verá que não só o jogador 1 coleta todos passageiros, mas também planeja quando reabastecer.

Outras constantes globais que você pode/deve editar ao longo deste EP são:

- `MAX_TURNS`: Determina o número máximo N de turnos que serão jogados (O jogo acaba no inicio do turno $N + 1$);
- `MAX_DEPTH`: Determina a profundidade máxima até a poda quando aplicável;
- `TANK_CAPACITY`: Determina o tamanho do tanque de combustível de ambos os agentes;[1]
- `DEFAULT_REFILL`: Quantidade de combustível que um agente coloca ao executar uma ação de `REFILL` dentro de um posto;
- `SHOW_INFO`: Se True, imprime a versão completa do jogo no terminal, incluindo o sumário final. Se False, imprime apenas o score final;
- `SHOW_DISPLAY`: Se True, abre a interface gráfica para reprodução do jogo, se False o jogo só executa no terminal.

Sua tarefa:

Sua tarefa nessa parte é implementar um agente planejador que tenha coma meta coletar todos os passageiros. Note que o agente deve assumir em sua formulação que o oponente permanece imóvel, mesmo que esse não seja o caso. Implemente seu agente na função `CollectAllAgent`, especificando o problema de busca em `CollectAllAgentProblem`.

O agente `CollectAllAgent` deve encontrar um plano ótimo realizando uma busca A* implementada no arquivo `util.py` (não modifique este arquivo). O custo de uma solução é a quantidade de combustível utilizada.

Você deve especificar o comportamento do agente `CollectAllAgent`, especificando a inicialização no método `__init__`, e a escolha de ações no método `get_action` (você pode criar métodos adicionais se desejar); O método `get_action` deve instanciar um problema de busca `CollectAllAgentProblem`, que você deve especificar, que deve ser resolvido usando o algoritmo de busca A* fornecido em `util.py`. Para isso você deve especificar uma heurística admissível a ser passada para a busca A*. Procure avaliar diversas heurísticas e escolher a que desempenha melhor. Sinta-se à vontade para se basear sua solução nos outros agentes planejadores fornecidos.

Teste sua implementação primeiramente contra um oponente imóvel `DoNothingAgent`, de forma a depurar eventuais erros de lógica ou programação.

Quando estiver confiante em sua implementação, teste seu agente contra um oponente que implementa o agente `GetClosestPersonOrRefillAgent`. Repare no efeito que o replanejamento tem.

Parte 2: Agentes Adversarias

A hipótese de um agente planejador que assume um oponente imóvel funciona razoavelmente devido ao replanejamento, mas ainda não consegue lidar com situações impostas pelo comportamento inesperado do oponente. Uma estratégia mais apurada (mas também mais custosa) é assumir um oponente adversarial escolhe suas ações de forma a minimizar nossa pontuação. Note que isso é equivalente a assumir um oponente que tentar maximizar sua própria pontuação apenas em um jogo de soma zero, o que não é o caso do jogo em questão.

No arquivo `ep2.py` é fornecida uma implementação do agente `AlphaBetaAgent`, que escolhe ações através do problema de busca adversarial com poda implementada em `AlphaBetaAgentProblem`. Para vê-lo em funcionamento, troque o agente do jogador 1 em `game.py`, altere a profundidade

[1] Por definição, todos os agentes começam com o tanque cheio e você pode supor que todos os carros tem o mesmo tamanho de tanque.

máxima da busca definida em `util.MAX_DEPTH` para 1 e coloque-o para jogar contra o oponente `GetClosestPersonOrRefillAgent`. Você verá que o tempo de processamento pode ser afetado, mas o agente até apresenta um resultado satisfatório. Entretanto, como a profundidade máxima da busca (`max_depth`=1) é muito pequena e a função de avaliação (`evaluation_function`) é muito simples, o agente ainda terá dificuldades contra outros agentes ou em tabuleiros mais complexos.

Avalie o efeito da profundidade máxima de busca na qualidade das ações escolhidas e no tempo de processamento, modificando o valor da constante `util.MAX_DEPTH` para ver se o comportamento do agente melhora (não recomendamos ultrapassar profundidades de 7 devido ao alto custo computacional).

Sua tarefa:

Sua tarefa nessa parte é fornecer uma função de avaliação que melhore consideravelmente o desempenho do agente `AlphaBetaAgent` (medido por sua pontuação final), quando jogando contra um oponente que implementa o agente `GetClosestPersonOrRefillAgent` (mas seu problema de busca adversarial não deve assumir esse comportamento do agente). Você também deve decidir o valor de profundidade máxima que forneça um bom compromisso entre a qualidade das ações e o tempo de processamento.

Implemente sua função de avaliação em `my_better_evaluation_function` (para que você possa compará-la com a função em `evaluation_function`).

Após implementá-la, modifique o seguinte trecho de código no `ep2.AlphaBetaAgentProblem`:

```
self.eval_fn = kwargs.get('eval_fn', self.evaluation_function)
```

Por:

```
self.eval_fn = kwargs.get('eval_fn', self.my_better_evaluation_function)
```

Talvez seja preciso testar o desempenho do agente em outros tabuleiros maiores e com diferentes configurações, já que o tabuleiro dado é relativamente simples.

Instruções para entrega

Você deve submeter via Paca apenas o arquivo `ep2.py` contendo a sua solução até às 23:55 do dia 01/05/2019. Para evitar que seu EP seja zerado, certifique-se que o arquivo foi submetido sem problemas, baixando e executando o arquivo do site. O arquivo deve conter um programa escrito para a versão 3.X do Python.