Ludwig-Maximilians-Universität München
Institut für Informatik
Lehrstuhl für Mobile und Verteilte Systeme
Prof. Dr. Claudia Linnhoff-Popien

mobile and distributed systems group

Praktikum Autonome Systeme
Wintersemester 2020/21
Übungsblatt 1 – Basics

Ziel der heutigen Praxisveranstaltung ist die Implementierung von einfachen Monte Carlo Search Algorithmen, mit denen ein Agent ein einfaches Navigationsproblem lösen soll.

Aufgabe 0:

Wir empfehlen die Nutzung von Conda (bzw. miniconda, siehe https://docs.conda.io/en/latest/miniconda.html) - einem Python Paket- und Umgebungsmanager.

Nach der Installation kann ein Conda Environment erzeugt und aktiviert werden mit:

```
$ conda create -n autonome python=3.8
$ source activate autonome
```

Die für diese Übung notwendigen python Libraries können dann installiert werden mit:

```
(autonome) $ pip install gym matplotlib seaborn moviepy
```

Denken Sie daran, dass das Conda environment immer zuerst aktiviert werden muss, wenn Sie ein neues Terminal öffnen:

```
$ source activate autonome
```

Sie erkennen, welches Conda environment aktiv ist, durch den vorgestellten environment Namen vor dem $-Zeichen. Der Übungscode kann dann ausgeführt werden mit:

```
(autonome) $ python main.py
```

Aufgabe 1: *Rooms* Domäne und Random Agent

Im ZIP-Archiv `autonome-systeme-uebung1.zip` befinden sich die Python Quellcode-Dateien zur Implementierung von autonomen Agenten, die in einer Navigationsdomäne (s. Abbildung 1) ein Ziel erreichen müssen.

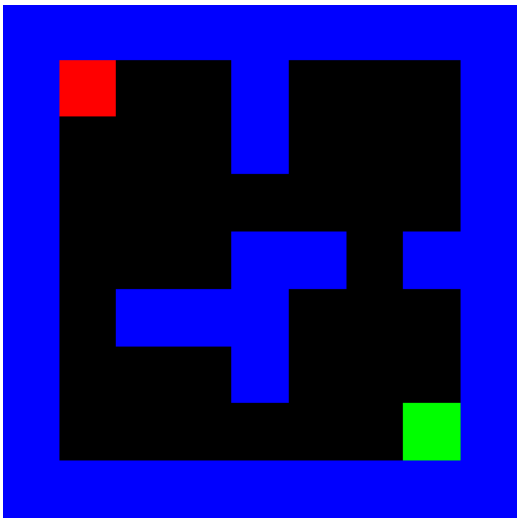

Abbildung 1: Rooms Domäne - Der Agent (rot) muss sich durch verschiedene Räume bis zu einem Ziel (grün) navigieren.

In `rooms.py` ist die Domäne `RoomsEnv` implementiert, welche mit der Funktion `load_env` instantiiert wird. Die *Rooms* Domäne erlaubt vier Aktionen (NORTH = 0, SOUTH = 1, WEST = 2, EAST = 3), mit denen sich der Agent in alle Himmelsrichtungen bewegen kann. Alle Aktionen werden durch Integer-Werte repräsentiert. Wenn der Agent versucht gegen eine Wand (blaue Gridzellen in Abbildung 1) zu gehen, geschieht nichts. Eine *Episode* d.h. ein Ablauf von Aktionsfolgen endet, wenn der Agent das Ziel erreicht hat, oder 100 Zeitschritte vergangen sind.

In `agent.py` befindet sich die Klasse `Agent`, welche als Grundgerüst für die zu implementierenden Agenten dient. Die Methode `policy` wählt eine Aktion zu einem bestimmten Zustand aus. Die Methode `update`, integriert Erfahrung in den internen Zustand des Agenten (für dieses Übungsblatt kann `update` ignoriert werden). Die Klasse `RandomAgent` liefert ein Beispiel für die Implementierung von `policy`.

In `main.py` wird eine Instanz der *Rooms* Domäne und des `RandomAgent` erzeugt, welcher in mehreren Episoden (s. Funktion `episode`) versucht, die *Rooms* Domäne zu lösen. Die Ergebnisse werden geplottet und die letzte Episode wird als Video in `rooms.mp4` gespeichert.

Starten Sie `main.py` und beobachten Sie das Verhalten von `RandomAgent` anhand des Plots und des Videos für 10 Episoden. Wie hoch ist die durchschnittliche Performance des Agenten?

Aufgabe 2: Monte Carlo Rollouts

Implementieren Sie einen Monte Carlo Rollout Planner. Orientieren Sie sich am Gerüst der Klasse `MonteCarloRolloutPlanner`.

In der `policy`-Methode werden die Variablen `Q_values` und `action_counts` erzeugt, um den erwarteten Return jeder verfügbaren Aktion zu schätzen. Zudem gibt es eine For-Schleife, welche `K = self.simulations` Mal durchlaufen wird. Die Variable `generative_model`, die zu Beginn eines Schleifendurchlaufs angelegt wird, repräsentiert den Simulator, mit dem geplant werden soll (Planen Sie nicht mit der echten Domäne `self.env`!).

Ihre Aufgabe ist es, die Planungslogik innerhalb der Schleife (an der Stelle # TODO Implement planning logic) zu implementieren.

Gehen Sie bei der Implementierung folgendermaßen vor:

1. Erzeugen Sie einen zufälligen Plan `random_plan` d.h. eine Sequenz der Länge $H =$ `self.horizon`, welche Zufallszahlen $\mathrm{a_t}$ von `0` bis `self.nr_actions-1` enthält.
2. Nutzen Sie den Simulator `generative_model`, um die Aktionen von `random_plan` durch zu simulieren. Falls Sie mit dem Interface von `RoomsEnv.step` noch nicht vertraut sind, orientieren Sie sich am Besten an der Funktion `episode` in `main.py`.
3. Berechnen Sie den `discounted_return` G_t von `random_plan`, indem Sie alle beobachteten Rewards $\mathcal{R}(\mathrm{s_t}, \mathrm{a_t})$ von 2. folgendermaßen addieren:

$$G_t = \sum_{t=0}^{H-1} \gamma^t \mathcal{R}(\mathrm{s_t}, \mathrm{a_t})$$

4. Aktualisieren Sie den erwarteten Return der ersten Aktion in `random_plan`, indem Sie den Wert `Q_values[random_plan[0]]` aktualisieren. **Hinweis:** Vergessen Sie <u>nicht</u> `action_counts[random_plan[0]]` zu inkrementieren da es sich bei den Werten in `Q_values` um Durchschnittswerte handelt.
5. Am Ende der `policy`-Methode wird die Aktion mit dem höchsten geschätzten erwarteten Return ausgewählt (diese Zeile ist bereits im Code enthalten).

Lassen Sie den `MonteCarloRolloutPlanner` über mehrere *Rooms* Episoden für verschiedene `K` und `H` laufen. Wie verhält sich die Performance bzgl. `K` und `H`?

Zusatz: Implementieren Sie zusätzlich eine Variante des `MonteCarloRolloutPlanner` welche nicht die Aktion mit dem höchsten Durchschnitts-Return, sondern die erste Aktion des besten Plans (also des Plans mit dem höchsten gesampleten `discounted_return`) auswählt.

Vergleichen Sie beide Planning Varianten über mehrere *Rooms* Episoden für verschiedene `K` und `H`. Wie verhält sich die Performance der beiden Varianten?