MAS Mandate 2

Keshav Goyal and Sooraj Sathish

March 2024

1 Recap of Learning Goal

1. **Problem Statement:**
   - A pre-defined graph $G(V, E)$ with goal state/node v_g.
2. **Objective:**
   - Find the shortest path from each node in the graph to the goal state.
3. **Model Solution:**
   - Utilizes a policy, which specifies the next action from each node/state in the graph.
   - Optimal policy leads from all states to the goal state through the shortest possible path.
4. **Proposed Algorithm:**
   - Agents start at random nodes in each episode.
   - All agents follow the same policy.
   - The policy is updated over episodes using replicator dynamics applied over pay-offs.
   - This process continues until an Evolutionary Stable Strategy (ESS) is reached.

2 Mandate 2 contributions

2.1 Designing the Payoff function

A trajectory is defined as the sequence of states (nodes) and actions (edges) from a start node to the goal node. When an agent experiences a trajectory that has a higher payoff than what is currently achieved by the policy, the policy is updated in the direction of the trajectory with a probability proportional to payoff advantage (replicator dynamics). For now, we abstract out the policy updating mechanism and focus on the payoff function which drives the update itself.

We identified three properties that the payoff function must show:

1. It should evaluate a trajectory based on the length (sum of edge weights) of the trajectory. Meaning, our payoff function should be dependent on the trajectory's **path length**.

2. The payoffs for trajectories with different starting states should be in a similar range irrespective of the start states. For example, if the shortest possible path from a node A is 50 while that from node B is 1000, a trivial payoff function only dependent on path length would always carry a bias for payoffs of trajectories originating from node B. To tackle this, our payoff function should depend on a **heuristic minimum** path length l from each node to the goal node. This heuristic is considered prior knowledge.

3. Payoffs should continuously shift depending on the best trajectory observed. For, example once a trajectory length of 50 is observed, the payoff of any trajectory with a length higher than 50 should now be lesser than before. To achieve this, the payoff function must be dependent on the **best trajectory** length achieved (h) from that start node. Initially, the best trajectory achieved is a **heuristic maximum** which is considered prior knowledge.

From the above properties, we get the following rules for the payoff function ($f(x,h,l)$, for trajectory length x).

1. $\frac{\partial f(x,h,l)}{\partial x} = -\lambda_1$ where $\lambda_1 > 0$ for any trajectory length x.
2. We set the following range: $0 \leq f \leq 1$. Meaning, that $f(l,h,l) = 1$ and $f(h,h,l) = 0$.
3. $\frac{\partial f(x,h,l)}{\partial h} = \lambda_2$ where $\lambda_2 > 0$. Meaning, that as the benchmark trajectory length (h) increases, the payoff for a particular x should increase as it is more and more better than h.

Then, the following process yields the expression for f.

$$f(x,h,l) = -\lambda_1.x$$

$$f(x,h,l) = \lambda_2.h$$

Adding,

$$2.f(x,h,l) = \lambda_2.h - \lambda_1.x$$

Substituting $f(l,h,l)$ and $f(h,h,l)$,

$$\lambda_1 = \frac{2}{h-l}$$

Substituting λ_1,

$$\lambda_2 = \frac{2}{h-l}$$

Thus, we get:

$$f(x,h,l) = \frac{h-x}{h-l}$$

However, for the practical implementation, we used a modification of this such that $f(h,h,l) = \eta$ where $\eta > 0$.

$$f(x,h,l) = \frac{h+\eta-x}{h+\eta-l}$$

In further sections, we discuss how the maximum and minimum heuristics are determined for implementation purposes.

2.2 Code base

We have coded a framework to execute the multi-agent path-finding system algorithm aimed at computing the shortest path. At this stage, we abstracted the policy implementation as a black box and proceeded to write the remaining essential code. This is because policy implementation requires knowledge of evolutionary game theory and replicator dynamics which will be covered in Mandate 3. The key components of the written code are encapsulated within the following primary Python files:

2.2.1 agent.py

- Defines the Agent class representing an agent navigating the graph.
- Implements methods for performing episodes and updating policies based on agent experiences.
- Algorithm 1 discusses how a single agent runs and stores an episode based on the policy. The episode runs until a goal state is reached or the maximum episode steps are taken. At each intermediate step, the action (next node) is chosen from the policy. The state, action, and cost (edge weight) are recorded as a tuple in the trajectory.
- Algorithm 2 discusses the update function which, given a trajectory, calculates payoffs and updates the policy. The agent treats each intermediate state of the trajectory as a starting state in a serialized manner and calculates the payoffs each time. This way we take maximum information from the trajectory and update policy for all intermediate states as well. After every episode, the update function is called to update the policy based on the trajectory of the episode.

Algorithm 1 Episode

```
1: Initialize: $\tau = ()$                          ▷ Initialise empty trajectory
2: Sample: $s_0$                                    ▷ Sample initial states
3: while $j < T$ do
4:     if $s_j \neq s_g$ then                       ▷ While agent has not reached the goal
5:         $a_j \leftarrow \pi(s_j)$                ▷ Sample action (next node) from policy
6:         $s_{j+1} \leftarrow a_j$                 ▷ Assign next state
7:         $w_j \leftarrow EdgeWeight(s_j, s_{j+1})$    ▷ Get traversed edge weight
8:         $\tau \leftarrow \tau || (s_j, a_j, w_j)$    ▷ Append $(s_j, a_j, w_j)$ tuple to trajectory
9:     end if
10: end while
```

Algorithm 2 Update

```
1: Initialize: $x = 0$                              ▷ Initialize total path length
2: for $\tau[i]$ in reversed($\tau$) do
3:     $s \leftarrow \tau[i][0]$                    ▷ $s$ is the new start state
4:     $a \leftarrow \tau[i][1]$                    ▷ next neighbour in trajectory
5:     $w \leftarrow \tau[i][2]$                    ▷ weight of chosen edge
6:     $x \leftarrow x + w$                         ▷ Total path length from $s$ to $s_g$
7:     $h \leftarrow max\_heuristics(s)$
8:     $l \leftarrow min\_heuristics(s)$
9:     if $x \leq h$ then
10:        payoff $\leftarrow f(x, h, l)$
11:        $max\_heuristics(s) \leftarrow x$        ▷ Update maximum as $x$ better than $h$
12:        policy.update($s, a$,payoff)
13:    end if
14: end for
```

2.2.2 mas.py

- Defines the MAS class responsible for managing multiple agents navigating a graph.

- Utilizes threading to run agents concurrently in separate threads. The number of agents can be set by the user. When all the agents have finished running episodes and performing corresponding updates to the policy, the MAS process ends. The policy accessed by the agents is common. In essence, changes made to the policy by an agent immediately affect the decisions of other agents.

- Another functionality is testing the obtained policy by evaluating agent performance against optimal paths. MAS calculates the distances to the goal state from all the nodes in the graph when navigating using the policy. While the optimal paths are calculated using the Dijkstra algorithm. The policy paths and optimal paths are then compared to evaluate the performance of a policy.

2.2.3 graph.py

- Implements the Graph class for generating and managing a random graph. Given the number of nodes and edges and a range for edge weights, it can produce a random graph with a randomly chosen goal state.

- **Minimum Heuristic:** To define a minimum heuristic, we first calculate the Dijkstra optimal paths from all nodes to the goal state. We then proceed to add random noise to these path lengths to obtain an implementation-focused minimum path length heuristic from each node to the goal state.

- **Maximum Heuristic:** This does not require a complex algorithmic process. The initial maximum possible path length is set as the total number of nodes multiplied by the maximum possible edge weight. This is trivially the maximum possible path length between any two nodes of the graph. This data keeps getting updated (Algorithm 2) as newer and better trajectories are witnessed.

2.2.4 main.py

- Imports necessary classes and sets up global parameters.

- Initializes a random graph, a policy, and a multi-agent system.

- Trains the multi-agent system by running agents for a specified number of epochs and episodes.

- Tests the trained policy by comparing distances against optimal paths.