Flatland Challenge with Graph Neural Networks

The official page of the challenge, with all the details and materials, can be reached here: https://www.aicrowd.com/challenges/flatland

Observation

The rail environment can be seen as composed of track sections, where each track section is a portion of the track delimited by 2 switches (forks). A switch is a cell where the train can choose between different direcitions, so it's a choice point. Other cells where there is only one possible direction are part of at least one track section. There are cells belonging to 2 track sections, these are the intersection cells. The state is the subjective point of view of an agent. Each agent has its own state-representation because the features are determined by relative attributes, such as the distance to the goal, the number of agents on the same track section navigating in the same or opposite direction and so on. The state is composed of the neighboring tracks features, up to a certain depth/level. In this way we perform GNN on a subset of the overall graph, determined by the depth of aggregation.

Track section features

This feature is used to represent railway tracks. In this approach, tracks become the nodes of the graph, while the switches are implicitly represented by the edges of the graph (possible directions from one node/track to another). In this way we simplify an important aspect: • Adjacent tracks which are not directly reachable with an action are not considered, because we only add an edge between 2 tracks that are directly linked and connected.

A possible problem emerging from this type of representation would be that intersecting paths are not directly represented. We can thus integrate this type of information in the track feature representation.

Computing observation

We want to optimize computation of observations only when it's needed, i.e. before making a decision. We update the dictionary AGENT*REQUIRED*OBS to tell the ObservationBuilder for which agent to compute obs. We compute observations only in these cases: 1. Agent is entering switch (obs for last cell of current path): we need obs to evaluate which path to take next 2. Agent is exiting a switch (obs for new cell of new path): we compute the obs because we could immediately meet another switch (track section only has 1 cell), so we need the observation in buffer 3. Agent is about to finish: we compute obs to save the experience tuple

Graph Neural Network (GNN) approach

The problem of finding a suitable observation is the variable size of the state. In fact by choosing a fixed-size state representation we could limit ourselves when changing the size of the rail environment. By using a fixed-size observation we use are determining a fixed architecture of the NN, which we would tend to implement as big as possible in order to capture the most of information available. But this leads to an increase in complexity. By using Graph Neural Networks we leverage the natural undelying graph structure of a railway. More importantly however, we don't limit ourselves to a fixed size state, since with a GNN we can have whatever number of nodes at a certain layer, because in the end all the values will be aggregated. The inductive bias of graph convolutions can be useful when generalizing on unseen and arbitrarly sized environments. We assign to each track section (node of the graph) a value. When navigating, each agent runs GNN independently from other agents (each agent has its own graph representation). Each agent access information about other agents position and speed. Each node represents the value of reaching it. So when the agent arrives at a switch, it has to decide which direction to go, and it does it by choosing the direction leading to the track section with the highest value. This is done by confronting all the values of the nodes which are directly reachable from the current switch. It also captures in some way the value of the possible path through it (similarly to Q-value in RL).

GNN propagation

The input layer is just the node representation (part of the state). We then compute the intermediary hidden states with NN and aggregations. We can start with 3 layers. The output value is computed not only for the track where the agent is (to determine if the best action is to STOP at the current track or not), but also for the tracks reachable from the next fork (choice points). For example another train could be passing in the chosen track, so we don't want to go against it and cause a deadlock We do this because we adopt a value-based approach, such that we choose the track with the highest value. Another approach uses Graph Attention Network instead of a custom network.

Reinforcement learning

Problems with policy-based methos: neighbors are order invariant, so each node can't recall from which direction a certain information comes from. We can't really determine the value of choosing a certain direction in this way. So, policy-based approach should be NOT FEASIBIBLE combined with Graph Neural Network approach. For this reason we use a Value-based approach where we only compute the values of the reachable track section from a certain switch and then select the path leading to the highest value.

GNN Convolution layer

This is the explaination of the VRSPConvolution.py file. The GNN convolution layer consists of a target-to-source convolutions, where the target track section (N2) features and the parent track section (N2, from which we reach the child track section) are concatenated into a 1D vector and then fed into a 2-layer Neural Network. The output of the first conv layer is part of the hidden representation of parent track section N2. In fact we compute the output for all the pairs (N2, N_child) and then combine then in a max, mean and min pooling layer. In this way we obtain a final hidden representation at layer 1 for node N2. This is done for all nodes at layer 1 (layer 0 is represented by input features, i.e. track features). We repeat the conv layer other 2 times for a total of 3 conv layers, so the depth of out GNN can be considered to be 3. This means that from the current node or track section where we are, we can reach the information of nodes 3 hops away from us. The root node is the only node about which we care the value, because it represents all the path reachable (at depth 3) from it. So, from the current switch, we consider as root node of different trees each reachable track section, in addition to the section we are at.

File browsing

- "train.py": is the main training file where all the experiences to be saved in the replay memory are computed. This file is quite messy, but the main purpose is to store the experience for an agent when it computes a decision for at a switch, when it reaches the target and when it's stuck in a deadlock. When at a switch, we save the observation at the current track section and the observation at the next track section (decided by the network). When done, we save the observation at the last switch encountered and the observation of the track section where the target is. When in a deadlock, we do the same as for a done agent. When the agent decides to stop at a switch, we also compute the observation at the current track, whereas the next state is also the same observation 1 timestep later. If a track section can be reached through more paths at a switch (usually happens then the switch is composed of more cells, so they are considered as 1 unique switch), all the possible paths for the same track section are considered as different nodes. This should help when a certain path is blocked by another agent, so an alternative one could be chosen.

• "dueling*double*dqn.py": contains the DQN agent implemented as DQN (planning to extend to Rainbow).

• "graph*for*observation.py": all the computation for the observation (also a mess, I don't suggest you to look at it). The function "*compute*node_observation()" gives you an idea of the features i computed for each track.

• "VRSPConv.py": the convolution layer of the GNN model. Pytorch geometric is the framework used to implement conv layer.

• "model.py": GNN model