Algorithm 1: Training Process for PPO_Train Algorithm

function ppo_train ($env, ppoTrainer, statePrepareList, greedyScores$);
Input: Environment inheritance from gym.Env, ppoTrainer object, list of all statePrepare for every problem and greedyScores list for problems
Data: statePrepares ← np.array(statePrepareList)
Data: State greedyScores ← np.array(greedyScores)
Data: best_score ← 0.0; n_steps ← 0
Data: score_history ← [ ]; remain_cap_history ← [ ]
for *Each i in N_TRAIN_STEP* **do**
  Data: batchs ← ppoTrainer.generate_batch(PROBLEMS_NUM, MAIN_BATCH_SIZE)
  for *Each batch in batchs* **do**
    env.setStatePrepare(statePrepares[batch])
    externalObservation, _ ← env.reset()
    Data: d
    one ← False
    while *not done* **do**
      Data: internalObservatio, actions, accepted_acctions, sumProbs, sumVals, sumRewards, steps ← ppoTrainer.make_steps(externalObservation, env.statePrepares)
      Data: externalObservation_, externalReward, done, info ← env.step(accepted_acctions)
      ppoTrainer.save_step(externalObservation, internalObservatio, actions, sumProbs, sumVals, sumRewards, steps, done)
      n_steps ← n_steps + 1
      if *n_steps % ppoTrainer.config.internal_batch == 0* **then**
        ppoTrainer.train_minibatch()
      end
      externalObservation ← externalObservation_
    end
    Data: scores, remain_cap_ratios ← env.final_score()
    Data: batch_score_per_grredy ← mean of scores/greedyScores[batch]
    score_history.append(batch_score_per_grredy)
    remain_cap_history.append(np.mean(remain_cap_ratios))
    Data: avg_score ← mean of score_history[-50:]
    if *avg_score > best_score* **then**
      Data: b
      est_score ← avg_score ppoTrainer.save_models()
    end
  end
end

Algorithm 2: make_step method in PPOTrainer class

function make_step ($externalObservation, statePrepares$);
Input: externalObservation, batch of statePrepares
Output: internalObservation, actions, accepted_actions, sumProbs, values, sumRewards, steps
Data: actions ← zero tensor with shape of $([MAIN\_BATCH\_SIZE, 0, 2])$
Data: sumLogProbs ← zero tensor with shape of $([MAIN\_BATCH\_SIZE])$
Data: sumRewards ←zero tensor with shape of $([MAIN\_BATCH\_SIZE])$
Data: internalObservation ← zero tensor with shape of $([MAIN\_BATCH\_SIZE, 0, generat\_link\_number + 1, input\_decode\_dim])$
Data: step ← tensor of $([1]MAIN\_BATCH\_SIZE)$
Data: steps ← zero tensor with shape of$((MAIN_BATCH_SIZE, 0))$
Data: prompt ← None
Data: accepted_actions ← numpy array of $([[[-1]2]generat\_link\_number]MAIN\_BATCH\_SIZE$
for *Each i in generat_link_number* **do**
    generatedInstance, generatedKnapsack, prompt ← actor_model.generateOneStep(step, externalObservation, prompt)
    Result: updated prompt if prompt == None and get new distributions as generatedInstance, generatedKnapsack
    act, log_prob ← _choose_actions(generatedInstance, generatedKnapsack)
    Result: get tensor of act as $([inst_act, ks_act])$ and log_pro as summation of instance log prob with knapsack log prob
    actions ← concatenate of actions and act
    reward ← reward(act, accepted_actions, step, prompt, statePrepares)
    Result: get internalReward and if instance is allocated in the knapsack update accepted_actions, step, prompt
    steps ← concatenate of steps, step to trac of step in actor model
    internalObservation ← concatenate of internalObservation, prompt
    sumProbs ← $sumProbs + prob$
    sumRewards ← $sumRewards + reward$
end

Algorithm 3: train_minibatch method in PPOTrainer class

```
function train_minibatch ();
for Each _ in ppo_epochs) do
    Data: batchs ← generate_batch()
    for Each index in MAIN_BATCH_SIZE do
        Data: obs ← memoryObs[index]
        Data: intObs ← memoryIntObs[index]
        Data: acts ← memoryAct[index]
        Data: probs ← memoryPrb[index]
        Data: rewards ← memoryRwd[index]
        Data: vals ← memoryVal[index]
        Data: stps ← memoryStp[index]
        Data: done ← memoryDon[index]
        Data: advantage ←zero tensor with shape of ([internal_batch])
        for Each t in internal_batch-1 do
            Data: discount ← 1
            Data: a_t ← 0
            for Each k between t and internal_batch-1 do
                a_t += discount * (rewards[k] + config.gamma * vals[k+1] * (1 - int(done[k]))
                 - vals[k])
                discount *= gamma * gae_lambda
            end
            advantage[t] ← a_t;
        end
        for Each batch in batches do
            Data: batchObs ← obs[batch]
            Data: batchIntObs ← intObs[batch]
            Data: batchActs ← acts[batch]
            Data: batchSteps ← stps[batch]
            Data: batchProbs ← probs[batch].to(device)
            Data: batchVals ← vals[batch].to(device)
            new_log_probs ← torch.tensor([0] * config.ppo_batch_size, dtype=torch.float64,
             device=device);
            for Each i in generat_link_number do
                generatedInstance, generatedKnapsack, _ ←
                        actor_model.generateOneStep(batchSteps[:,i], batchObs.to(device),
                        batchIntObs[:,i]) Result: get new distributions as generatedInstance,
                        generatedKnapsack for external and internal observations
                Data: inst_dist ← Categorical(generatedInstance)
                Data: ks_dist ← Categorical(generatedKnapsack)
                Result: get new torch distributions categorical objects
                Data: inst_log_probs ← inst_dist.log_prob(batchActs[:,i,0])
                Data: ks_log_probs ← ks_dist.log_prob(batchActs[:,i,1])
                Data: newProbs ← inst_log_probs + ks_log_probs
                new_log_probs += newProbs
            end
            Data: newVal ← critic_model(batchObs)
            Data: prob_ratio ← exponental of (new_log_probs - batchProbs)
            Data: weighted_probs ← advantage[batch] * prob_ratio
            Data: weighted_clipped_probs ← torch.clamp(prob_ratio, 1 - cliprange, 1 +
                    cliprange) * advantage[batch]
            Data: actor_loss ← mean of -min between weighted_probs and
                    weighted_clipped_probs
            Data: returns ← advantage[batch] + batchVals
            Data: critic_loss ← mean of euclidean distance between returns and newVal
            Data: total_loss ← actor_loss + 0.5 * critic_loss
            actor_optimizer.zero_grad()
            critic_optimizer.zero_grad()
            total_loss.backward()
            actor_optimizer.step()
            critic_optimizer.step()
        end
    end
end
```