# A APPENDIX

## A.1 ALGORITHMS

We tested the combination of TBV with two planning algorithms: Monte Carlo Tree Search (equipped with some auxiliary mechanisms) and Best First Search. Here we present the pseudo-code for both algorithms and provide some additional explanation.

### A.1.1 BESTFS

In Algorithm 3 we present the pseudocode for BestFS algorithm used in out experiments. In Data Structures 4 and 5 we present two additional data structures used by our implementation of BesFS.

**Algorithm 3** BestFS planner

**Require:** $model$ learned model used in Algorithm 1
$V$ value function
$C$ number of nodes expansions per step
$\gamma$ discount factor
**Use:** $F$ priority queue of unexpanded nodes (fringe)
$G$ graph of all seen nodes

**function** RESET
  $F \leftarrow \emptyset$
  $G \leftarrow \emptyset$
**function** CHOOSE_ACTION(`state`)
  $G$.ADD($state$)
  **if** $state \notin G$ **then**
    $state.n\_visits \leftarrow 1$
  **else**
    $state.n\_visits \leftarrow state.n\_visits + 1$
  $reachable \leftarrow$ FIND_REACHABLE_NODES($G, state$) ▷ find all nodes reachable from state
  EXPAND_GRAPH($reachable$)
  $best\_node \leftarrow G$.FIND BEST NODE($reachable$) ▷ see Algorithm 5
  $path \leftarrow$ FIND_PATH_TO_NODE($best\_node$)
  **return** $path.first\_action$
**function** EXPAND_GRAPH(reachable)
  **for** $1 \dots C$ **do**
    $n \leftarrow F$.POP_BEST_NODE($reachable$) ▷ see Algorithm 4
    EXPAND_GRAPH_NODE($n$)
**function** EXPAND_GRAPH_NODE(`n`, `reachable`)
  $children \leftarrow []$
  **for** $a \in \mathcal{A}$ **do**
    $model$.LOAD_STATE($n.state$)
    $new\_state \leftarrow model$.STEP($a$)
    $new\_state.uncertainity \leftarrow \max_{action} model.uncertainity$(`new_state`, `action`)
    **if** $new\_state \notin G$ **then**
      $value \leftarrow V$.EVALUATE($new\_state$)
      $new\_state$.SET_VALUE()
      $F$.ADD_NEW_NODE($new\_state$)
      $G$.ADD_NEW_NODE($new\_state$)
      $reachable$.ADD($new\_state$)
  $F$.REMOVE($n$) ▷ fringe contains only unexpanded nodes
**function** GET_GRAPH_OF_PLANNING()
  **return** $G$

**Data structure 4** Fringe (priority queue)

**Require:** $S_F$ ▷ set of nodes

**function** ADD_NODE(`n`)
    $S_F$.ADD($n$)

**function** REMOVE(`n`)
    $S_F$.REMOVE($n$)

**function** POP_BEST_NODE(`reachable`)
    $nodes \leftarrow S_F$.INTERSECTION($reachable$)
    **return** $\arg\max_{n \in nodes} n.uncertainity$

**Data structure 5** BestFS graph

**Require:** $S_G$ ▷ set of nodes

**function** ADD_NODE(`n`)
    $S_G$.ADD($n$)

**function** LEXICOGRAPHICAL_MAX(nodes) ▷ returns max according to keys: (solved, -n_visits, uncertainity)

**function** FIND_BEST_NODE(`reachable`, `state`)
    $nodes \leftarrow S$.INTERSECTION($reachable$)
    **return** LEXICOGRAPHICAL_MAX($nodes$)

### A.1.2 MCTS WITH AVOID HISTORY COEFFICIENT DEAD ENDS AND AVOID SELF LOOPS

In Algorithm 6 we present the general structure of our MCTS planner. In Algorithms 7, 10, 8 and 9 we present our version of key functions in MCTS. In Algorithm 11 we provide pseudo code for ad auxiliary dead end mechanism used in this paper. Finally, in Data structure 12 we present the functionality of graph node used in this implementation.

**Algorithm 6** MCTS planner

| | | |
|---|---|---|
| **Require:** | $model$ | learned model used in Algorithm 1 |
| | $V$ | value function ensemble (assigns a vector of value estimates to state) |
| | $C$ | number of MCTS passes |
| | $\kappa$ | score factor for value uncertainty |
| | $\gamma$ | discount factor |
| | $\gamma_{dead}$ | dead end value |
| | $\gamma_{ah}$ | avoid history coefficient |
| **Use:** | $T$ | planning graph |
| | $S_{visited}$ | set of states visited by the agent |
| | $S_{seen}$ | set of states seen in one MCTS pass |

**function** RESET
    $T \leftarrow \emptyset$

**function** CHOOSE_ACTION(`state`)
    $root \leftarrow state$
    **for** $1 \ldots$ `C` **do**
        MCTS_PASS($state$)
    **return** $\arg\max_{a \in \mathcal{A}}(n.child(a).value)$

**function** MCTS_PASS(`root`)
    $S_{seen} \leftarrow \emptyset$ ▷ used in Algorithm 11
    $path, leaf \leftarrow$ TRAVERSAL($root$)
    $value \leftarrow$ EXPAND_LEAF($leaf, model$)
    BACKPROPAGATE($value, path$)

**function** GET_GRAPH_OF_PLANNING() **return** $T$

**Algorithm 7** `traversal()`

**Input:** `root`

`n` $\leftarrow$ `root`
`path` $\leftarrow \emptyset$
**while** `n` is not a leaf **do**
  $a \leftarrow$ `select_child`$(n)$
  **if** $a$ is `None` **then** ▷ dead end, terminal or leaf
    **break**
  $path$.APPEND$((n, a)$
  $n \leftarrow n.child(a)$
**return** $path, n$ ▷ `n` $\notin$ `path`

**Algorithm 8** `expand_leaf()`

**Require:** $\gamma_{dead}$, $V$, $model$
**Input:** `leaf` ▷ MCTS tree node without children

**if** $leaf$ is terminal **then**
  UPDATE$(leaf, 0.)$
  **return** $0$.
**else if** IS_DEAD_END$(leaf)$ **then**
  UPDATE$(leaf, \gamma_{dead})$
  **return** $\gamma_{dead}$
**else**
  **for** $a \in \mathcal{A}$ **do**
    $new\_node \leftarrow$ CREATE_NODE()
    $model$.LOAD_STATE$(leaf.state)$
    $new\_state \leftarrow model$.STEP$(a)$
    $new\_node.state \leftarrow new\_state$
    **if** $new\_state$ not yet visited **then**
      $new\_state.value \leftarrow V(new\_state)$
    $leaf.child(a) \leftarrow new\_node$
**return** $leaf.value$

**Algorithm 9** `backpropagate()`

**Require:** $\gamma$
**Input:** $v$, $path$

**for** $(\mathtt{n}, a)$ in `reversed`$(path)$ **do**
  $v \leftarrow n.reward + \gamma v$
  $n$.UPDATE$(v)$

**Algorithm 10** `select_child()`

**Input:** `n`

**if** $n$ not expanded or terminal **then**
  **return** `None`
**if** IS_DEAD_END$(n)$ **then**
  **return** `None`
**else**
  $\mathcal{A}_{allowed} \leftarrow$ ALLOWED$(n)$
  **return** $\arg\max_{a \in \mathcal{A}_{allowed}} n.child(a)$.EVAL()

**Algorithm 11** Dead end detection

**Require:** $S_{seen}$
**Input:** $v$, $path$

**function** ALLOWED(n)
  $\mathcal{A}_{allowed} \leftarrow \emptyset$
  **for** $a \in \mathcal{A}$ **do**
    **if** $n.child(a) \notin S_{seen}$ **then**
      $\mathcal{A}_{allowed}$.ADD$(a)$
  **return** $\mathcal{A}_{allowed}$
**function** IS_DEAD_END(n)
  $\mathcal{A}_{allowed} \leftarrow$ ALLOWED$(n)$
  **if** $\mathcal{A}_{allowed} \neq \emptyset$ **then**
    **return** `False`
  **else**
    **return** `True`

**Data structure 12** `node`

**Attributes:**

| | |
|---|---|
| $count$ | MCTS node counter |
| $value$ | value ensemble estimate |
| $visits$ | number of agents visits |
| $child$ | list of children nodes |

**function** UPDATE(value)
  $self.value \leftarrow self.value + value$
  $self.count \leftarrow self.count + 1$
**function** ADD_VISIT()
  $self.visits \leftarrow self.visits + 1$
**function** EVAL()
  $s \leftarrow$ SCORE$(self.value, self.count)$
  $std \leftarrow$ STD$(self.value)$ ▷ uncertainty of value ensemble
  **return** $score + \gamma_{ah} self.visits + \kappa std$

## A.2 TRAINING SETUP

Code for all our experiments can be accessed at `https://github.com/ComradeMisha/TrustButVerify`.

Our experiments adhere to the general model-based training loop logic, described in Algorithm 1. We use a distributed system with 32 workers solving distinct episodes, where data gathered across a batch of workers is collected in two common experience buffers: the replay buffer for trainable model of size 50000 and the replay buffer for value function of size 30000. Before the solving of actual episode, we collect 1000 random trajectories, that we use for initial training of the model of

environment. This initial training has critical importance for the performance of our agents. Episodes are limited to: 600 steps for ToyMontezumaRevenge and 1000 steps for the Tower of Hanoi.

To update value network for MCTS we follow approach of Milos et al. (2019) (similar to Hamrick et al. (2020)) for BestFS experiments we simply calculate future discounted returns for each transition encountered by agent.

## A.3 Network architectures for model and value

### A.3.1 Single networks

In every experiments we use two neural network architectures: one to estimate the value function of a given state and the other to predict the outcome of taking a step in the environment. In ToyMR we represent the state as a vector of length 17 and in the Tower of Hanoi as a vector of length $3 \times$ number of discs (it is due to one-hot encoding of the peg number for each disc).

In both environments, for value estimation we use an MLP (multilayer perceptron) architecture with two hidden layers of 50 neurons and ReLu non-linearity. For model, in both environments, we use an MLP with four hidden layers of 250 neurons and ReLU activation functions.

The model network has three outputs: the difference between next and current observations (delta target), reward and the episode termination flag (0 or 1).

### A.3.2 Ensemble

For both value estimation and model we use ensemble of networks with architectures described in Section A.3.1. The usage of ensembles is however different for value network and model network. Ensemble of model network is used as described in Algorithm 2. In case of value estimation, the final result of value is an average of predictions across ensemble. The standard deviation of this predictions multiplied by $\kappa$ is used by MCTS as an auxiliary score added to each vertex in the search tree.

In Algorithm 13 we present the procedure of transforming the output of ensemble of model networks to a valid prediction of environment signal. In our experiments, it turned out to be beneficial to train the model to predict change between observations rather then the ready observations. The transformation performs the following step: collects predictions of all networks in ensemble, then averages next predicted change of observations, predicted rewards and predicted end of episode flags. Averaged observation change is added to current state, then clipped to the range of possible values (some coordinates of the state vector are binary variables), rounded to integers and returned as the next predicted state. Reward and end of episode flag are predicted to be true if their average value across networks in ensemble is larger than $0.5$.

To stabilize performance of value and model ensembles we used additional masks mechanism. We create some number of networks (given by ensemble size parameter), that are trained with the same data buffer, but for prediction each worker uses only a random subset of given size (given by number of masks parameter).

## A.4 Quantile Rank value analysis

Figure 8 presents performance of BestFS Agent in the Tower of Hanoi environment for different values of Quantile Rank. We observed the the best performance was for values between 0.8 and 0.95, and when the parameter was within the appropriate range of values, the performance was not very sensitive to the changes of quantile rank parameter. We observed similar behaviour in ToyMR experiments - the best performance was for quantile rank between 0.9 and 0.95 but it was hard to narrow down this interval, due to aforementioned little sensitivity.

**Algorithm 13** Transforming model ensemble predictions

**function** PREDICT_STEP(`state`, `action`)
  $next\_observations \leftarrow []$
  $rewards \leftarrow []$
  $is\_done\_flags \leftarrow []$
  **for** $network \in ensemble$ **do**:
    $(next\_obs\_delta, reward, is\_done) \leftarrow network.\text{PREDICT}(state, action)$
    $next\_obs \leftarrow state + next\_obs\_delta$
    $next\_observations.\text{APPEND}(next\_obs)$
    $rewards.\text{APPEND}(reward)$
    $is\_done\_flags.\text{APPEND}(is\_done)$
  $predicted\_reward \leftarrow \text{AVERAGE}(next\_reward)$
  **if** $predicted\_reward > 0.5$ **then**
    $predicted\_reward \leftarrow 1$
  **else**
    $predicted\_reward \leftarrow 0$
  $predicted\_is\_done \leftarrow \text{AVERAGE}(is\_done\_flags)$
  **if** $predicted\_is\_done > 0.5$ **then**
    $predicted\_is\_done \leftarrow True$
  **else**
    $predicted\_is\_done \leftarrow False$
  $predicted\_next\_obs \leftarrow \text{AVERAGE}(next\_observations)$
  $predicted\_next\_obs \leftarrow \text{CLIP}(predicted\_next\_obs)$
  $predicted\_next\_obs \leftarrow \text{ROUND}(predicted\_next\_obs)$
  **return**$(predicted\_next\_obs, predicted\_reward\, predicted\_is\_done)$

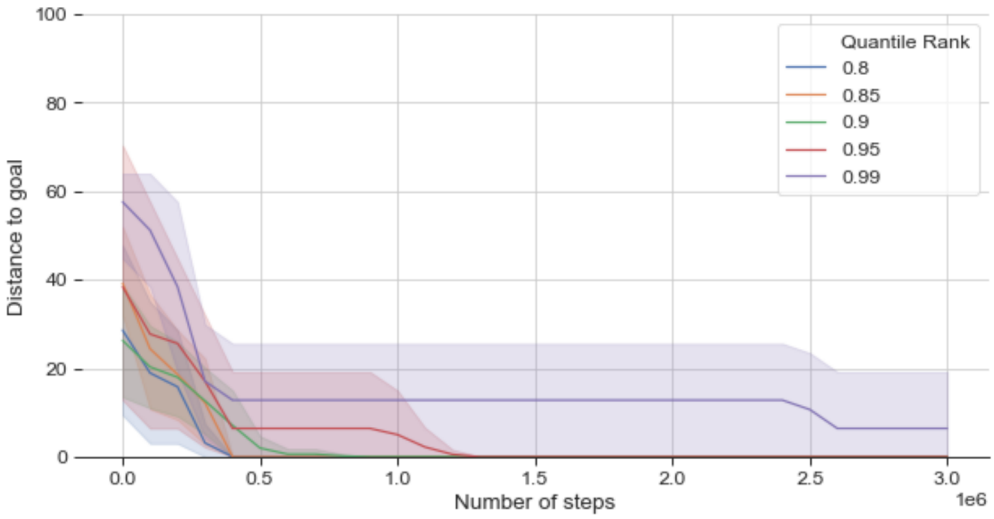


Figure 8: Performance of *TBV* BestFS on the Tower of Hanoi domain with different values of Qantile Rank

## A.5 Random TBV rejection analysis

In our preliminary experiments we have seen that without random rejection of TBV mechanism the agents often tends to get stuck in cycles with high ensemble disagreement and is unable to leave such cycle until the end of episode (since model is not updated in during the episode). In Figure 9 we present the performance in Tower of Hanoi for different values of frequency of random TBV

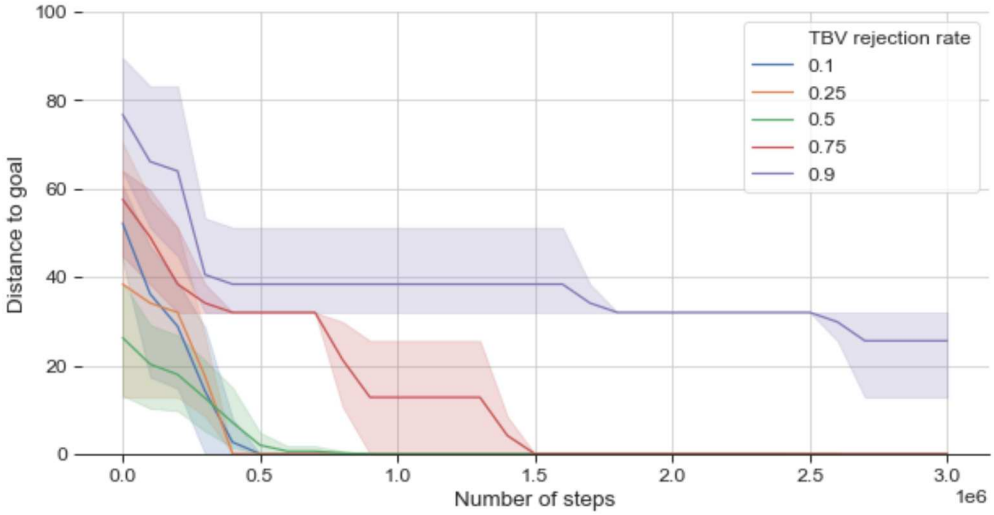


Figure 9: Performance of *TBV* BestFS on the Tower of Hanoi domain with different frequencies of random rejection of TBV

rejection (while the rest of parameters is same as in our best experiments). It can be seen, that if the frequency of TBV override (i.e. frequency with which we allow TBV to change planners action) is between 0.5 and 0.9 we obtained best results. Also, TBV is not very sensitive to this parameter as long it is in an appropriate range of values.

### A.6 HYPER-PARAMETERS

In tables 1 and 2 we present hyper-parameters used in our experiments.

| Parameter | Toy MR BestFS | Toy MR MCTS | Tower of Hanoi BestFS | Tower of Hanoi MCTS |
|---|---|---|---|---|
| Number of planner passes $C$ | 10 | 10 | 10 | 10 |
| Discounting factor $\gamma$ | 0.99 | 0.99 | 0.99 | 0.99 |
| $\varepsilon$ greedy exploration for baslines | 0.001 | 0.02 | 0.001 | 0.02 |
| score factor for value uncertainty $\kappa$ | - | 3 | - | 3 |
| Dead end value | - | $-0.2$ | - | $-0.2$ |
| Avoid history coefficient | - | $-0.2$ | - | $-0.2$ |
| Value ensemble size [2] | 20 | 20 | 20 | 20 |
| Value ensemble mask size [2] | 10 | 10 | 10 | 10 |
| Model ensemble size [2] | 8 | 8 | 8 | 8 |
| Model ensemble mask size [2] | 4 | 4 | 4 | 4 |
| Optimizer | RMSprop | RMSprop | RMSprop | RMSprop |
| Learning rate | 2.5e$-$4 | 2.5e$-$4 | 2.5e$-$4 | 2.5e$-$4 |
| Batch size for value training | 32 | 32 | 32 | 32 |
| Batch size for model training | 1024 | 1024 | 1024 | 1024 |

[2] See Section A.3.2 for explanation of these parameters.

Table 1: Hyper-parameters values used in our experiments.

For MCTS we took hyperparameters from Milos et al. (2019). As they used setup without learned model we needed to tune model architecture and model training parameters on ToyMontezumaRe-

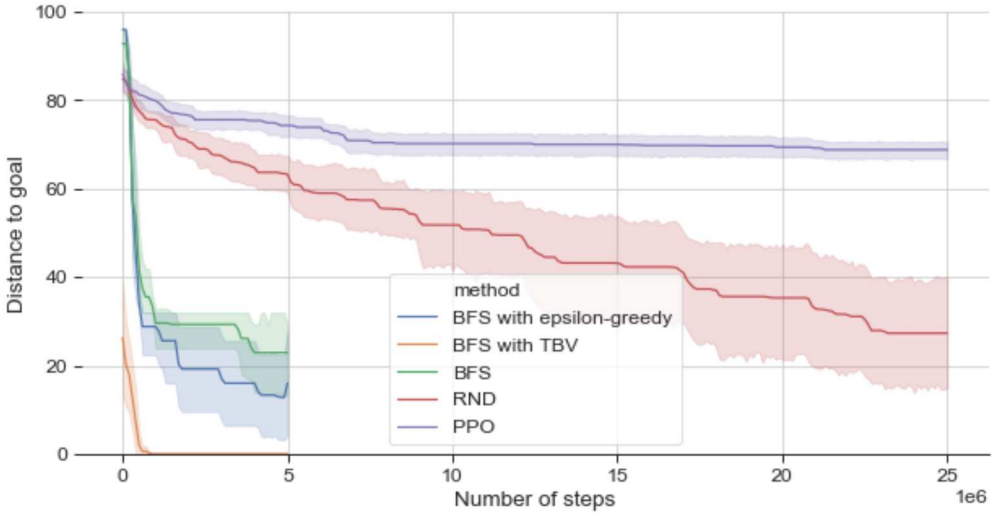


Figure 10: Performance of RND and PPO on Tower of Hanoi with longer training.

venge domain. Then, we separately tuned epsilon and Quantile Rank for each combination of domain/planner.

## A.7 RND AND PPO

RND continued to further explore state space for Hanoi (figure 10), for comparison we also show performance of PPO (the same parameters as RND but without intrinsic reward).

| Parameter | ToyMR | Tower of Hanoi |
|---|---|---|
| Rollout length | 128 | 128 |
| Maximal length of an episode | 600 | 1000 |
| Total number of rollouts per environment | 6200 | 6200 |
| Number of minibatches | 4 | 4 |
| Number of optimization epochs | 16 | 4 |
| Coefficient of extrinsic reward | 1 | 1 |
| Coefficient of intrinsic reward | 100 | 10 |
| Number of parallel environments | 32 | 32 |
| Learning rate | 0.001 | 0.001 |
| Optimization algorithm | Adam | Adam |
| $\lambda$ | 0.95 | 0.95 |
| Entropy coefficient | 0.001 | 0.001 |
| Proportion of experience used for training predictor | 1.0 | 1.0 |
| $\gamma_E$ | 0.999 | 0.999 |
| $\gamma_I$ | 0.99 | 0.99 |
| Clip range | [0.9, 1.1] | [0.9, 1.1] |
| Policy architecture | FCN | FCN |

Table 2: Default hyper-parameters for PPO and RND algorithms for ToyMR and Tower of Hanoi experiments.

| Parameter | Value |
|---|---|
| Number of optimization epochs | [1, 4, 16] |
| Coefficient of intrinsic reward | [1, 3, 10, 30, 100, 300] |
| Learning rate | $[5 \cdot 10^{-2}, 10^{-2}, 5 \cdot 10^{-3}, 10^{-3}, 5 \cdot 10^{-4}, 10^{-4}]$ |
| $\lambda$ | $[0.95, 0.99]$ |
| Proportion of experience used for training predictor | [0.25, 1.0] |
| $\gamma_E$ | $[0.999, 0.9999]$ |
| $\gamma_I$ | $[0.99, 0.999]$ |
| Policy architecture layers number | $[2, 3, 4]$ |
| Policy architecture layers width | $[64, 128, 256]$ |
| Random target and prediction networks last layer width | $[64, 128]$ |

Table 3: Hyper-parameters for RND algorithm checked during tuning process of ToyMontezumaRevenge.