Supplementary materials for "What is the model in model-based planning?"

November 6, 2020

1 Appendix A: Task representation implementation details

Integrating the four task representations and the MBP framework we have described in this work required a few small additional considerations. The baseline representation, for example, included object positions in its compressed representation and thus new task instances with slightly different object positions produced compressed representations that did not match any previously observed states. In this case, there would be no entry in the agent's lookup table for this state, and the agent would be unable predict about what might happen next. When this happened, our agents simply selected randomly from the currently available actions.

Secondly, to ensure that the behavioral differences between agents were solely due to their task representations, we wanted each agent's transition function to be as close as possible to what would have been learned under an ideal learning process. For the rule-based representation, this simply required providing hand-coded rules that appropriately captured the dynamics of each task. For the three tabular representations, however, it would have been impractical to try and fill-in the hundreds or thousands of observed state transitions that an agent would have stored during training. Instead we provided the tabular agents with a simple update process that compressed each state s, action a, and subsequent state s' observed during training into compressed representations $(C_S(s), C_A(a), C_S(s'))$. We then updated the agent's table entry for $(C_S(s), C_A(a))$ to keep track of $C_S(s')$ and $n_{C_S(s')}$ where $n_{C_S(s')}$ was the number of times that the compressed format $C_S(s')$ was visited from taking $C_A(a)$ in $C_S(s)$. When then used these counts to estimate $P(s'|s, a)$ for each agent.

Thirdly, one challenge of using this planning architecture was that MCTS tries to explore all actions from each state at least once before exploring any action multiple times. Given that the rule-based agent selected from object instances rather than categories of objects and that some states in our task contained dozens of instances of wall objects, the rule-based agent required too many simulations to produce reasonable value estimates. Thus, we added a heuristic to our planner to ignore any wall objects in order to make the action space more tractable. To allow fair comparison, we provided the same heuristic to the informative-object-features and relational-categories representations when we

```
start -
    ("START", segment):                 p_0
segment -
    (trap, ):                           p_1
    (treasure, ):                       p_2
trap -
    ("AVOID_TRAP", "END_SUBGOAL"):      p_3
treasure -
    ("TOUCH_TREASURE", "END_SUBGOAL"):  p_4
```

Figure 1: A probabilistic context-free grammar for generating solutions to the Object Number Variation Task.

evaluated them. This introduced another difficulty: the probability of selecting a wall object for these agents became 0. Since our human participants did occasionally select wall objects as targets this caused a problem when fitting our agent parameters to human data. To account for this, we added a small probability ϵ of selecting a random target object. This ϵ parameter was set to the base rate of wall selection in human participants (roughly 3% of object interactions).

2 Appendix B: Task generation

To measure generalization across within-domain variation, we needed to generate many instances of each task that varied along a given dimension. To this end, we used a generative process to produce task instances by creating solutions and then laying out objects on a grid to make that solution attainable.

For example, instances of the Object Number Variation Task involve the player navigating to treasures while avoiding traps and then heading to an exit. We can represent the commonalities in these puzzles using a generative grammar. We used the probabilistic, context-free grammar (PCFG) shown in Figure 1 to generate solution phrases like the following:

$$\texttt{TOUCH}(\text{treasure}_1) \rightarrow \texttt{END}(\text{goal}_1) \rightarrow \texttt{AVOID}(\text{trap}_1) \rightarrow \texttt{END}(\text{goal}_2) \rightarrow \ldots \rightarrow \texttt{TOUCH}(\text{exit}_1)$$

Depending on the parameters of the PCFG, each solution produced by this grammar involved a different number of treasures, traps, and walls.

We then used the algorithm shown in Figure 2 to arrange the solution objects on a grid.

The process described in Figure 2 ensures each sub-goal in the solution can be reached without necessarily going through any other sub-goal. In the ONVT for example, this meant that all treasures could be touched without first touching a trap. In the OFVT we ensured that it was always possible to reach the exit without going through doors. This prevented players who did not learn how keys work from getting stuck. In the OCVT, we ensured that there was always one treasure and trap available without using teleporters or conveyor belts. This allowed players to learn what each object type did before having

1. Initialize a starting point in space.

2. If the current item is not an "end-sub-goal" item, either:

   (i) Add an empty space with some probability p_e. Place wall objects in all unoccupied spaces around the current cell.

   (ii) Add the current item with probability $1 - p_e$ and remove the current item from the solution. Place wall objects in all unoccupied spaces around the current cell.

3. If the first item in the solution is an "end-sub-goal" item, select a previously-placed wall at random to use as the new current cell and remove the wall in that space. Remove the current item from the solution.

4. Select a random space that is adjacent to the current cell and filled with a wall that does not border any other filled cells. Remove the wall and set this as the current cell. If no such cell exists, remove everything generated since the last non-empty item and try again. If that doesn't work, keep removing segments until some maximum number of attempts is reached.

5. Repeat 2-4 until there are no more items in the solution phrase.

Figure 2: Task instance generation process

to figure out how to use the teleporters or belts. We also ensured that each conveyor belt/teleporter setup could be interacted with independently. Splitting each task instance into sub-goals allowed us to control planning complexity by limiting the number of sub-goals available in each instance. This in turn allowed us to minimize how much working memory, attention, or general planning ability our tasks required.

Once we had grid layouts for each solution we randomly assigned shapes and colors to each object. For each participant, we used random self-avoiding walks in 2D grids to generate unique shapes for each distinct kind of object. We then generated unique colors by representing colors as HSL (Hue, Saturation, Light) vectors and selecting n_k different hue values that were equidistant from each other in HSL space, where n_k was the number of object kinds that needed distinct colors for a given task. On average participants saw 4-7 different shapes and colors depending on which specific task they played (although the specific set of shapes and colors used varied from participant to participant). The same setup could be used to generate instances with dozens of unique shapes and colors for training artificial agents that require more variation.

This shape and color randomization process also allowed us to ensure that each instance was unique along the key dimension of variation for each task. All other dimensions were held constant within participants across instances. Thus, the treasures in the ONVT would be different shapes and colors for each participant, but would remain the same shape and color from instance to instance within participants. This allowed us to at least partially account for the prior knowledge brought to each task (e.g., 'red' is bad and 'green' is good). Generating many different shapes and colors introduced the possibility that a task representation might fail during evlaution due to encountering shapes/colors that were not seen during training. Thus we set up our instance generating process to ensure that every color used in the evaluation instances had been previously observed in at least one training instance. In other words, while an evaluation instance might have involved never-before-seen green walls, the agent was guaranteed to have seen other green objects during training, just never green objects that were acting as walls.

3 Appendix C: Model specification and simulation

We explored four different potential task representations in this work. Algorithms 1-3 provide pseudocode for the MCTS + Sarsa(λ) value estimation process used by all four model classes, while Table 1 includes a description of all the parameters used by these algorithms and their fitted values.

Algorithm 1 shows the central planning loop used by all four agents, wherein the agents selected actions based on the current task state and then updated their internal model according to the outcomes of that action. Here, the *GetState* function returned the appropriate state representation for each model. Some actions lead to multi-step animations, as in the Object Composition Variation Task where touching a conveyor belt caused the agent to move along a fixed path. While on this path, the *HasAnimation* function would return TRUE, and the *GetNextAnimationAction* could be used to identify the next animated action. After completing the appropriate action, agents attempted to update their task representation based on the new state and any reward earned. For tabular agents, *UpdateModel* added $(C_S(s), C_A(a), C_S(s'))$ to the agent's tabular transition function and $(C_S(s), C_A(a), r)$ to the tabular reward function. For rule-based agents, there was no update step, as the rules of the task did not change during play.

Algorithm 1 Planning with MCTS and Sarsa(λ)

```
Initialize $M_c, w_c$
$s \leftarrow GetState()$
$i \leftarrow 0$
while $GetGameState(s) = 0$ and $i < M_c$ do
    $i \leftarrow i + 1$
    if $HasAnimation(s) = TRUE$ then
        $a \leftarrow GetNextAnimationAction(s)$
    else
        $a \leftarrow GetNextMove(s)$
    end if
    $s' \leftarrow GetState()$
    $r \leftarrow GetReward(w_c)$
    $UpdateModel(s, a, s', r)$
    $s \leftarrow s'$
end while
```

Algorithm 2 shows the action selection process, which involved running simulations, using those simulations to produce value estimates, and then stochastically selecting actions based on their estimated values. To find the set of currently available actions, *GetAction* used a BFS algorithm to find the set of all objects that could be currently reached by the agent without passing through any other objects. Actions were randomly selected using a softmax function, parameterized by τ.

Algorithm 2 GetNextMove(s_0)

```
Initialize $\tau$
$\mathbf{a}_0 \leftarrow GetActions(s)$
$\mathbf{V} \leftarrow RunSimulations(\mathbf{a}_0, s_0)$
$\mathbf{w} \leftarrow []$
for $a \in \mathbf{a}_0$ do
    $\mathbf{w}[a] \leftarrow exp(\frac{V[s_0,a]}{\tau})$
end for
$\mathbf{w} \leftarrow \frac{\mathbf{w}}{\sum \mathbf{w}}$
$a \leftarrow Random(\mathbf{a}_0, \mathbf{w})$
return $a$
```

Finally, Algorithm 3 produced value estimates for all currently available actions by using MCTS and internal transition and reward functions to generate simulated episodes and Sarsa(λ) to estimate values based on these episodes. Here *GetNextStates* used the model's internal transition and reward functions to simulate the effect of taking each action in **a** from state s, while *NormalizeRewards* normalized the rewards from each state to a 0-1 range (see Vodopivec, Samothrakis, and Ster (2017) for more details about why normalizing the rewards within each state helps the value estimation process converge more quickly). After normalization, an the algorithm calculated upper confidence bounds for all available actions based on the current number of times that each state/action pair has been visited during simulation. These visitation counts were reset at the beginning of each value estimation process by the *ResetVisits* function. Each agent then selected an action based on this upper confidence bound and simulated the effect of that action using its internal transition and reward functions. For tabular agents, *TransitionFunction* and *RewardFunction* looked up compressed representations s and a and then drew a predicted next state or reward with probability related to the proportion of times that next state had been visited (or reward had been obtained) from $(C_S(s), C_A(a))$ in the past. For the rule based agent, the agent's logic engine was used to sequentially apply all known rules to the current state and action and produce a new state, which included reward information as part of the state description. Once all episodes had been generated, value estimates were produced using typical Sarsa(λ) update rules.

Algorithm 3 RunSimulations($\mathbf{a}_0, s_0$)

```
Initialize $\alpha, \gamma, \lambda, D_s, N_s, V_0, w_c$
$ResetVisits()$
$\mathbf{V} \leftarrow []$
for $n = 1 \rightarrow n = N_s$ do
    $\mathbf{E} \leftarrow []$
    $s \leftarrow s_0$
    $\mathbf{a} \leftarrow \mathbf{a}_0$
    for $d = 1 \rightarrow d = D_s$ do
        $\mathbf{s}, \mathbf{r} \leftarrow GetNextStates(s, \mathbf{a})$
        $\mathbf{r} \leftarrow NormalizeRewards(\mathbf{r}, s)$
        $\mathbf{UCB} \leftarrow \mathbf{r} + \sqrt{2 * log(\frac{\sum_{\mathbf{a}} N_v}{N_v})}$
        if $N_{v,a} = 0$ for any $a$ then
            $a \leftarrow Random(UnvisitedActions(s))$
        else
            $a \leftarrow ArgMax(\mathbf{UCB})$
        end if
        if $(s, a) \notin \mathbf{V}$ then
            $\mathbf{V}[s, a] \leftarrow V_0$
        end if
        $s' \leftarrow TransitionFunction(s, a)$
        $r \leftarrow RewardFunction(s, a, w_c)$
        $UpdateEpisode(\mathbf{E}, (s, a, r, s'))$
        $s \leftarrow s'$
        $\mathbf{a} \leftarrow GetActions(s)$
    end for
    $\delta_{sum} \leftarrow 0$
    $V_{next} \leftarrow 0$
    for $s, a, r, s' \in Reversed(\mathbf{E})$ do
        $\delta \leftarrow r + \gamma * V_{next} - \mathbf{V}[s, a]$
        $\delta_{sum} \leftarrow \lambda * \gamma * \delta_{sum} + \delta$
        if $(s, a) \in \mathbf{V}$ then
            $\mathbf{V}[s, a] \leftarrow \alpha * \delta_{sum} + \mathbf{V}[s, a]$
            $V_{next} \leftarrow \mathbf{V}[s, a]$
        else
            $V_{next} \leftarrow V_0$
        end if
    end for
end for
return $\mathbf{V}$
```

Parameter	Description	Range	Baseline	Inf.-obj.-features	Rel.-categories	Rule-based
α	Learning rate	0.001 - 1.0	0.001	0.001	0.5	0.001
γ	Discount rate	0.001 - 1.0	1.0	1.0	0.5	1.0
λ	Trace decay rate	0.001 - 1.0	0.001	0.5	1.0	0.5
D_s	Max. action depth	N/A	5	5	5	5
M_c	Max. actions per task	N/A	325 (directional actions)	15 (target objects)	15	15
N_s	Number of simulations	5 - 30	10	10	10	20
τ	Action selection temperature	-8.0 - 0.0	0.0	0.0	0.0	0.0
V_0	Initial value estimate for unexplored actions	N/A	0	0	0	0
w_c	Rel. weight of additional actions	0.001 - 1.0	0.001	0.001	0.5	0.001

Table 1: Summary of key model parameters for Algorithms 1-3. Range column shows ranges used during performance and LL fitting processes. Variables marked N/A were not fit. Right-most four columns represent performance-maximizing values for each of the four model classes.

References

Vodopivec, T., Samothrakis, S., & Ster, B. (2017). On monte carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, *60*, 881-936.