Contents in Appendices:

- In Appendix A, we describe each of the components in GATA in detail.
- In Appendix B, we provide detailed information on how we pre-train GATA's graph updater with the two proposed methods (i.e., OG and COC).
- In Appendix C, we provide detailed information on GATA-GTP, the discrete version of GATA. Since the action scorer module is the same as in GATA, this appendix elaborates on how a discrete graph updater works and how to pre-train the discrete graph updater.
- In Appendix D, we provide additional results and discussions. This includes training curves, training scores, testing scores, and high-res examples of the belief graphs learned by GATA. We provide a set of probing experiments to show that the belief graphs learned by GATA can capture useful information for relation classification tasks. We also provide qualitative analysis on GATA's OG task, which also suggests the belief graphs contain useful information for reconstructing the text observation O_t.
- In Appendix E, we provide implementation details for all our experiments.
- In Appendix F, we show examples of graphs in TextWorld games.

A Details of GATA

Notations

In this section, we use O_t to denote text observation at game step t, C_t to denote a list of action candidate provided by a game, and $\mathcal{G}_t$ to denote a belief graph that represents GATA's belief to the state.

We use L to refer to a linear transformation and L^f means it is followed by a non-linear activation function f. Brackets $[\cdot;\cdot]$ denote vector concatenation. Overall structure of GATA is shown in Figure 2.

A.1 Graph Encoder

As briefly mentioned in Section 3.3, GATA utilizes a graph encoder which is based on R-GCN [32].

To better leverage information from relation labels, when computing each node's representation, we also condition it on a relation representation E:

$$\tilde{h}_i = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r} W_r^l[h_j^l; E_r] + W_0^l[h_i^l; E_r]\right), \tag{5}$$

in which, l denotes the l-th layer of the R-GCN, $\mathcal{N}_i^r$ denotes the set of neighbor indices of node i under relation $r \in \mathcal{R}$, $\mathcal{R}$ indicates the set of different relations, W_r^l and W_0^l are trainable parameters. Since we use continuous graphs, $\mathcal{N}_i^r$ includes all nodes (including node i itself). To stabilize the model and preventing from the potential explosion introduced by stacking R-GCNs with continuous graphs, we use Tanh function as σ (in contrast with the commonly used ReLU function).

As the initial input h^0 to the graph encoder, we concatenate a node embedding vector and the averaged word embeddings of node names. Similarly, for each relation r, E_r is the concatenation of a relation embedding vector and the averaged word embeddings of r's label. Both node embedding and relation embedding vectors are randomly initialized and trainable.

To further help our graph encoder to learn with multiple layers of R-GCN, we add highway connections [36] between layers:

$$\begin{aligned} g &= L^{\text{sigmoid}}(\tilde{h}_i), \\ h_i^{l+1} &= g \odot \tilde{h}_i + (1-g) \odot h_i^l, \end{aligned} \tag{6}$$

where $\odot$ indicates element-wise multiplication.

We use a 6-layer graph encoder, with a hidden size H of 64 in each layer. The node embedding size is 100, relation embedding size is 32. The number of bases we use is 3.

A.2 Text Encoder

We use a transformer-based text encoder, which consists of a word embedding layer and a transformer block [43]. Specifically, word embeddings are initialized by the 300-dimensional fastText [28] word vectors trained on Common Crawl (600B tokens) and kept fixed during training in all settings.

The transformer block consists of a stack of 5 convolutional layers, a self-attention layer, and a 2-layer MLP with a ReLU non-linear activation function in between. In the block, each convolutional layer has 64 filters, each kernel's size is 5. In the self-attention layer, we use a block hidden size H of 64, as well as a single head attention mechanism. Layernorm [6] is applied after each component inside the block. Following standard transformer training, we add positional encodings into each block's input.

We use the same text encoder to process text observation O_t and the action candidate list C_t. The resulting representations are $h_{O_t} \in \mathbb{R}^{L_{O_t} \times H}$ and $h_{C_t} \in \mathbb{R}^{N_{C_t} \times L_{C_t} \times H}$, where L_{O_t} is the number of tokens in O_t, N_{C_t} denotes the number of action candidates provided, L_{C_t} denotes the maximum number of tokens in C_t, and $H = 64$ is the hidden size.

A.3 Representation Aggregator

The representation aggregator aims to combine the text observation representations and graph representations together. Therefore this module is activated only when both the text observation O_t and the graph input $\mathcal{G}_t$ are provided. In cases where either of them is absent, for instance, when training the agent with only $\mathcal{G}^{\text{belief}}$ as input, the aggregator will be deactivated and the graph representation will be directly fed into the scorer.

For simplicity, we omit the subscript t denoting game step in this subsection. At any game step, the graph encoder processes graph input $\mathcal{G}$, and generates the graph representation $h_{\mathcal{G}} \in \mathbb{R}^{N_{\mathcal{G}} \times H}$. The text encoder processes text observation O to generate text representation $h_O \in \mathbb{R}^{L_O \times H}$. $N_{\mathcal{G}}$ denotes the number of nodes in the graph $\mathcal{G}$, L_O denotes the number of tokens in O.

We adopt a standard representation aggregation method from question answering literature [49] to combine the two representations using attention mechanism.

Specifically, the aggregator first uses an MLP to convert both $h_{\mathcal{G}}$ and h_O into the same space, the resulting tensors are denoted as $h'_{\mathcal{G}} \in \mathbb{R}^{N_{\mathcal{G}} \times H}$ and $h'_O \in \mathbb{R}^{L_O \times H}$. Then, a trilinear similarity function [33] is used to compute the similarities between each token in h'_O with each node in $h'_{\mathcal{G}}$. The similarity between ith token in h'_O and jth node in $h'_{\mathcal{G}}$ is thus computed by:

$$\text{Sim}(i, j) = W(h'_{O_i}, h'_{\mathcal{G}_j}, h'_{O_i} \odot h'_{\mathcal{G}_j}), \tag{7}$$

where W is trainable parameters in the trilinear function. By applying the above computation for each pair of h'_O and $h'_{\mathcal{G}}$, a similarity matrix $S \in \mathbb{R}^{L_O \times N_{\mathcal{G}}}$ is resulted.

Softmax of the similarity matrix S along both dimensions (number of nodes $N_{\mathcal{G}}$ and number of tokens L_O) are computed, producing $S_{\mathcal{G}}$ and S_O. The information contained in the two representations are then aggregated by:

$$\begin{aligned} h_{O\mathcal{G}} &= [h'_O; P; h'_O \odot P; h'_O \odot Q], \\ P &= S_{\mathcal{G}} h'^{\top}_{\mathcal{G}}, \\ Q &= S_{\mathcal{G}} S_O^{\top} h'^{\top}_O, \end{aligned} \tag{8}$$

where $h_{O\mathcal{G}} \in \mathbb{R}^{L_O \times 4H}$ is the aggregated observation representation, each token in text is represented by the weighted sum of graph representations. Similarly, the aggregated graph representation $h_{\mathcal{G}O} \in \mathbb{R}^{N_{\mathcal{G}} \times 4H}$ can also be obtained, where each node in the graph is represented by the weighted sum of text representations. Finally, a linear transformation projects the two aggregated representations to a space with size H of 64:

$$\begin{aligned} h_{\mathcal{G}O} &= L(h_{\mathcal{G}O}), \\ h_{O\mathcal{G}} &= L(h_{O\mathcal{G}}). \end{aligned} \tag{9}$$

A.4 Scorer

The scorer consists of a self-attention layer, a masked mean pooling layer, and a two-layer MLP. As shown in Figure 2 and described above, the input to the scorer is the action candidate representation h_{C_t}, and one of the following game state representation:

$$s_t = \begin{cases} h_{\mathcal{G}_t} & \text{if only graph input is available,} \\ h_{O_t} & \text{if only text observation is available, this degrades GATA to a Tr-DQN,} \\ h_{\mathcal{G}O_t}, h_{O\mathcal{G}_t} & \text{if both are available.} \end{cases}$$

First, a self-attention is applied to the game state representation s_t, producing $\hat{s_t}$. If s_t includes graph representations, this self-attention mechanism will reinforce the connection between each node and its related nodes. Similarly, if s_t includes text representation, the self-attention mechanism strengthens the connection between each token and other related tokens. Further, masked mean pooling is applied to the self-attended state representation $\hat{s_t}$ and the action candidate representation h_{C_t}, this results in a state representation vector and a list of action candidate representation vectors. We then concatenate the resulting vectors and feed them into a 2-layer MLP with a ReLU non-linear activation function in between. The second MLP layer has an output dimension of 1, after squeezing the last dimension, the resulting vector is of size N_{C_t}, which is the number of action candidates provided at game step t. We use this vector as the score of each action candidate.

A.5 The f_Δ Function

As mentioned in Eqn. 2, f_Δ is an aggregator that combines information in $\mathcal{G}_{t-1}$, A_{t-1}, and O_t to generate the graph difference Δg_t.

In specific, f_Δ uses the same architecture as the representation aggregator described in Appendix A.3. Denoting the aggregator as a function Aggr:

$$h_{PQ}, h_{QP} = \mathrm{Aggr}(h_P, h_Q), \tag{10}$$

f_Δ takes text observation representations $h_{O_t} \in \mathbb{R}^{L_{O_t} \times H}$, belief graph representations $h_{\mathcal{G}_{t-1}} \in \mathbb{R}^{N_\mathcal{G} \times H}$, and action representations $h_{A_{t-1}} \in \mathbb{R}^{L_{A_{t-1}} \times H}$ as input. L_{O_t} and $L_{A_{t-1}}$ are the number of tokens in O_t and A_{t-1}, respectively; $N_\mathcal{G}$ is the number of nodes in the graph; H is hidden size of the input representations.

We first aggregate h_{O_t} with $h_{\mathcal{G}_{t-1}}$, then similarly $h_{A_{t-1}}$ with $h_{\mathcal{G}_{t-1}}$:

$$\begin{aligned} h_{O\mathcal{G}}, h_{\mathcal{G}O} &= \mathrm{Aggr}(h_{O_t}, h_{\mathcal{G}_{t-1}}), \\ h_{A\mathcal{G}}, h_{\mathcal{G}A} &= \mathrm{Aggr}(h_{A_{t-1}}, h_{\mathcal{G}_{t-1}}). \end{aligned} \tag{11}$$

The output of f_Δ is:

$$\Delta g_t = [\bar{h_{O\mathcal{G}}}; \bar{h_{\mathcal{G}O}}; \bar{h_{A\mathcal{G}}}; \bar{h_{\mathcal{G}A}}], \tag{12}$$

where $\bar{X}$ is the masked mean of X on the first dimension. The resulting concatenated vector Δg_t has the size of $\mathbb{R}^{4H}$.

A.6 The f_d Function

f_d is a decoder that maps a hidden graph representation $h_t \in \mathbb{R}^H$ (generated by the RNN) into a continuous adjacency tensor $\mathcal{G} \in [-1, 1]^{2\mathcal{R} \times \mathcal{N} \times \mathcal{N}}$.

Specifically, f_d consists of a 2-layer MLP:

$$\begin{aligned} h_1 &= L_1^{\mathrm{ReLU}}(h_t), \\ h_2 &= L_2^{\tanh}(h_1). \end{aligned} \tag{13}$$

In which, $h_1 \in \mathbb{R}^H$, $h_2 \in [-1, 1]^{\mathcal{R} \times \mathcal{N} \times \mathcal{N}}$. To better facilitate the message passing process of R-GCNs used in GATA's graph encoder, we explicitly use the transposed h_2 to represent the inversed relations in the belief graph. Thus, we have $\mathcal{G}$ defined as:

$$\mathcal{G} = [h_2; h_2^T]. \tag{14}$$

The transpose is performed on the last two dimensions (both of size $\mathcal{N}$), the concatenation is performed on the dimension of relations.

The tanh activation function on top of the second layer of the MLP restricts the range of our belief graph $\mathcal{G}$ within $[-1, 1]$. Empirically we find it helpful to keep the input of the multi-layer graph neural networks (the R-GCN graph encoder) in this range.

B Details of Pre-training Graph Updater for GATA

As briefly described in Section 3.2, we design two self-supervised tasks to pre-train the graph updater module of GATA. As training data, we gather a collection of transitions from the *FTWP* dataset. Here, we denote a transition as a 3-tuple (O_{t-1}, A_{t-1}, O_t). Specifically, given text observation O_{t-1}, an action A_{t-1} is issued; this leads to a new game state and O_t is returned from the game engine. Since the graph updater is recurrent (we use an RNN as its graph operation function), the set of transitions are stored in the order they are collected.

B.1 Observation Generation (OG)

As shown in Figure 4, given a transition (O_{t-1}, A_{t-1}, O_t), we use the belief graph $\mathcal{G}_t$ and A_{t-1} to reconstruct O_t. $\mathcal{G}_t$ is generated by the graph updater, conditioned on the recurrent information h_{t-1} carried over from previous data point in the transition sequence.

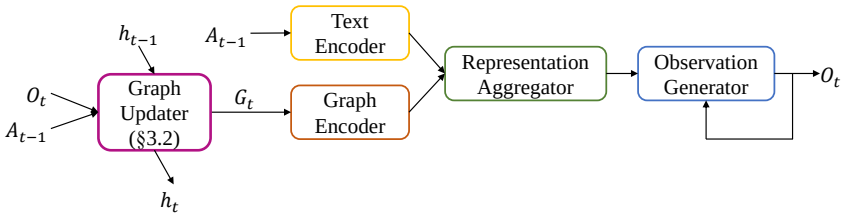


Figure 4: Observation generation model.

B.1.1 Observation Generator Layer

The observation generator is a transformer-based decoder. It consists of a word embedding layer, a transformer block, and a projection layer.

Similar to the text encoder, the embedding layer is frozen after initializing with the pre-trained fastText [28] word embeddings. Inside the transformer block, there is one self attention layer, two attention layers and a 3-layer MLP with ReLU non-linear activation functions in between. Taking word embedding vectors and the two aggregated representations produced by the representation aggregator as input, the self-attention layer first generates a contextual encoding vectors for the words. These vectors are then fed into the two attention layers to compute attention with graph representations and text observation representations respectively. The two resulting vectors are thus concatenated, and they are fed into the 3-layer MLP. The block hidden size of this transformer is $H = 64$.

Finally, the output of the transformer block is fed into the projection layer, which is a linear transformation with output size same as the vocabulary size. The resulting logits are then normalized by a softmax to generate a probability distribution over all words in vocabulary.

Following common practice, we also use a mask to prevent the decoder transformer to access "future" information during training.

B.2 Contrastive Observation Classification (COC)

The contrastive observation classification task shares the same goal of ensuring the generated belief graph $\mathcal{G}_t$ encodes the necessary information describing the environment state at step t. However, instead of generating O_t from $\mathcal{G}_t$, it requires a model to differentiate the real O_t from some $\tilde{O}_t$ that are randomly sampled from other data points. In this task, the belief graph does not need to encode

the syntactical information as in the observation generation task, rather, a model can use its full capacity to learn the semantic information of the current environmental state.

We illustrate our contrastive observation classification model in Figure 5. This model shares most components with the previously introduced observation generation model, except replacing the observation generator module by a discriminator.

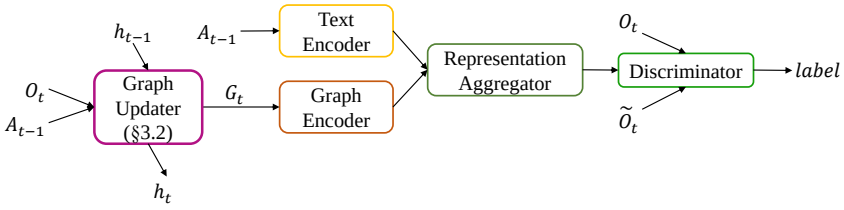


Figure 5: Contrastive observation classification model.

B.3 Reusing Graph Encoder in Action Scorer

Both of the graph updater and action selector modules rely heavily on the graph encoder layer. It is natural to reuse the graph updater's graph encoder during the RL training of action selector. Specifically, we use the pre-trained graph encoder (and all its dependencies such as node embeddings and relation embeddings) from either the above model to initialize the graph encoder in action selector. In such settings, we fine-tune the graph encoders during RL training. In Appendix D, we compare GATA's performance between reusing the graph encoders with randomly initialize them.

C GATA-GTP and Discrete Belief Graph

As mentioned in Section 3.4, since the TextWorld API provides ground-truth (discrete) KGs that describe game states at each step, we provide an agent that utilizes this information, as a strong baseline to GATA. To accommodate the discrete nature of KGs provided by TextWorld, we propose GATA-GTP, which has the same action scorer with GATA, but equipped with a discrete graph updater. We show the overview structure of GATA-GTP in Figure 6.

C.1 Discrete Graph Updater

In the discrete graph setting, we follow [53], updating $\mathcal{G}_t$ with a set of discrete update operations that act on $\mathcal{G}_{t-1}$. In particular, we model the (discrete) Δg_t as a set of update operations, wherein each update operation is a sequence of tokens. We define the following two elementary operations so that any graph update can be achieved in $k \geq 0$ such operations:

- `add(node1, node2, relation)`: add a directed edge, named `relation`, between `node1` and `node2`.
- `delete(node1, node2, relation)`: delete a directed edge, named `relation`, between `node1` and `node2`. If the edge does not exist, ignore this command.

Given a new observation string O_t and $\mathcal{G}_{t-1}$, the agent generates $k \geq 0$ such operations to merge the newly observed information into its belief graph.

Table 3: Update operations matching the transition in Figure 1.

```
<s> add player shed at <|> add shed backyard west_of <|> add wooden door shed
east_of <|> add toolbox shed in <|> add toolbox closed is <|> add workbench
shed in <|> delete player backyard at </s>
```

We formulate the update generation task as a sequence-to-sequence (Seq2Seq) problem and use a transformer-based model [43] to generate token sequences for the operations. We adopt the decoding strategy from [27], where given an observation sequence O_t and a belief graph $\mathcal{G}_{t-1}$, the agent

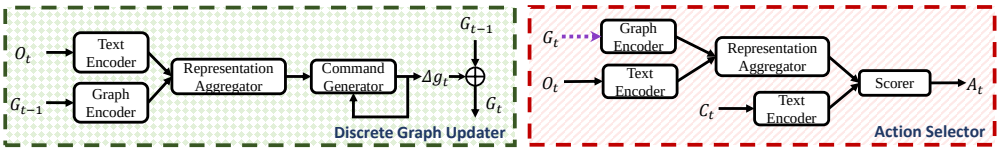


Figure 6: GATA-GTP in detail. The coloring scheme is same as in Figure 1. The discrete graph updater first generates Δg_t using $\mathcal{G}_{t-1}$ and O_t. Afterwards the action selector uses O_t and the updated graph $\mathcal{G}_t$ to select A_t from the list of action candidates C_t. Purple dotted line indicates a detached connection (i.e., no back-propagation through such connection).

generates a sequence of tokens that contains multiple graph update operations as subsequences, separated by a delimiter token `<|>`.

Since Seq2Seq set generation models are known to learn better with a consistent output ordering [27], we sort the ground-truth operations (e.g., always `add` before `delete`) for training. For the transition shown in Figure 1, the generated sequence is shown in Table 3.

C.2 Pre-training Discrete Graph Updater

As described above, we frame the discrete graph updating behavior as a language generation task. We denote this task as command generation (CG). Similar to the continuous version of graph updater in GATA, we pre-train the discrete graph updater using transitions collected from the *FTWP* dataset. It is worth mentioning that despite requiring ground-truth KGs in *FTWP* dataset, GATA-GTP does not require any ground-truth graph in the RL game to train and evaluate the action scorer.

For training discrete graph updater, we use the $\mathcal{G}^{\text{seen}}$ type of graphs provided by the TextWorld API. Specifically, at game step t, $\mathcal{G}_t^{\text{seen}}$ is a discrete partial KG that contains information the agent has observed from the beginning until step t. It is only possible to train an agent to generate belief about the world it has seen and experienced.

In the collection *FTWP* transitions, every data point contains two consecutive graphs, we convert the difference between the graphs to ground-truth update operations (i.e., `add` and `delete` commands). We use standard teacher forcing technique to train the transformer-based Seq2Seq model. Specifically, conditioned on the output of representation aggregator, the command generator is required to predict the k^{th} token of the target sequence given all the ground-truth tokens up to time step $k-1$. The command generator module is transformer-based decoder, similar to the observation generator described in Appendix B.1.1. Negative log-likelihood is used as loss function for optimization. An illustration of the command generation model is shown in Figure 7.

Figure 7: Command Generation Model.

During the RL training of action selector, the graph updater is detached without any back-propagation performed. It generates token-by-token started by a begin-of-sequence token, until it generates an end-of-sequence token, or hitting the maximum sequence length limit. The resulting tokens are consequently used to update the discrete belief graph.

C.3 Pre-training a Discrete Graph Encoder for Action Scorer

In the discrete graph setting, we take advantage of the accessibility of the ground-truth graphs. Therefore we also consider various pre-training approaches to improve the performance of the graph encoder in the action selection module. Similar to the training of graph updater, we use transitions collected from the *FTWP* dataset as training data.

In particular, here we define a transition as a 6-tuple $(\mathcal{G}_{t-1}, O_{t-1}, C_{t-1}, A_{t-1}, \mathcal{G}_t, O_t$. Specifically, given $\mathcal{G}_{t-1}$ and O_{t-1}, an action A_{t-1} is selected from the candidate list C_{t-1}; this leads to a new game state $\mathcal{S}_t$, thus $\mathcal{G}_t$ and O_t are returned. Note that $\mathcal{G}_t$ in transitions can either be $\mathcal{G}_t^{\text{full}}$ that describes the full environment state or $\mathcal{G}_t^{\text{seen}}$ that describes the part of state that the agent has experienced.

In this section, we start with providing details of the pre-training tasks and their corresponding models, and then show these models' performance for each of the tasks.

C.3.1 Action Prediction (AP)

Given a transition $(\mathcal{G}_{t-1}, O_{t-1}, C_{t-1}, A_{t-1}, \mathcal{G}_t, O_t, r_{t-1})$, we use A_{t-1} as positive example and use all other action candidates in C_{t-1} as negative examples. A model is required to identify A_{t-1} amongst all action candidates given two consecutive graphs $\mathcal{G}_{t-1}$ and $\mathcal{G}_t$.

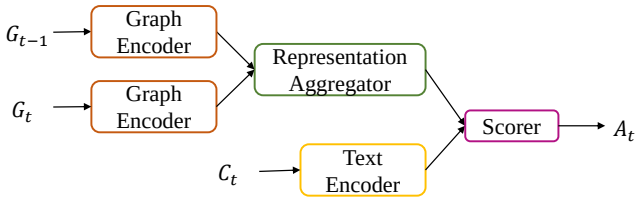


Figure 8: Action Prediction Model.

We use a model with similar structure and components as the action selector of GATA. As illustrated in Figure 8, the graph encoder first converts the two input graphs $\mathcal{G}_{t-1}$ and $\mathcal{G}_t$ into hidden representations, the representation aggregator combines them using attention mechanism. The list of action candidates (which includes A_{t-1} and all negative examples) are fed into the text encoder to generate action candidate representations. The scorer thus takes these representations and the aggregated graph representations as input, and it outputs a ranking over all action candidates.

In order to achieve good performance in this setting, the bi-directional attention between $\mathcal{G}_{t-1}$ and $\mathcal{G}_t$ in the representation aggregator needs to effectively determine the difference between the two sparse graphs. To achieve that, the graph encoder has to extract useful information since often the difference between $\mathcal{G}_{t-1}$ and $\mathcal{G}_t$ is minute (e.g., before and after taking an apple from the table, the only change is the location of the apple).

C.3.2 State Prediction (SP)

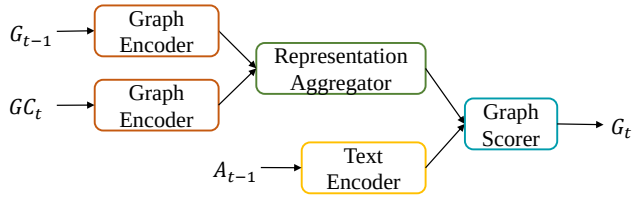


Figure 9: State Prediction Model.

Given a transition $(\mathcal{G}_{t-1}, O_{t-1}, C_{t-1}, A_{t-1}, \mathcal{G}_t, O_t, r_{t-1})$, we use $\mathcal{G}_t$ as positive example and gather a set of game states by issuing all other actions in C_{t-1} except A_{t-1}. We use the set of graphs representing the resulting game states as negative samples. In this task, a model is required to identify $\mathcal{G}_t$ amongst all graph candidates GC_t given the previous graph $\mathcal{G}_{t-1}$ and the action taken A_{t-1}.

As shown in Figure 9, a similar model is used to train both the SP and AP tasks.

C.3.3 Deep Graph Infomax (DGI)

This is inspired by Velickovic et al., [44]. Given a transition $(\mathcal{G}_{t-1}, O_{t-1}, C_{t-1}, A_{t-1}, \mathcal{G}_t, O_t, r_{t-1})$, we map the graph $\mathcal{G}_t$ into its node embedding space. The node embedding vectors of $\mathcal{G}_t$ is denoted as H. We randomly shuffle some of the node embedding vectors to construct a "corrupted" version of the node representations, denoted as $\tilde{H}$.

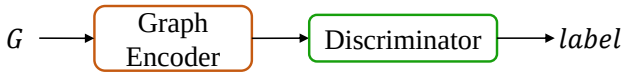


Figure 10: Deep Graph Infomax Model.

Given node representations $H = \{\overrightarrow{h_1}, \overrightarrow{h_2}, ..., \overrightarrow{h_N}\}$ and corrupted representations of these nodes $\tilde{H} = \{\overrightarrow{\tilde{h}_1}, \overrightarrow{\tilde{h}_2}, ..., \overrightarrow{\tilde{h}_N}\}$, where N is the number of vertices in the graph, a model is required to discriminate between the original and corrupted representations of nodes. As shown in Figure 10, the model is composed of a graph encoder and a discriminator. Specifically, following [44], we utilize a noise-contrastive objective with a binary cross-entropy (BCE) loss between the samples from the joint (positive examples) and the product of marginals (negative examples). To enable the discriminator to discriminate between $\mathcal{G}_t$ and the negative samples, the graph encoder must learn useful graph representations at both global and local level.

C.3.4 Performance on Graph Encoder Pre-training Tasks

We provide test performance of all the models described above for graph representation learning. We fine-tune the models on validation set and report their performance on test set.

Additionally, as mentioned in Section 3.3 and Appendix A, we adapt the original R-GCN to condition the graph representation on additional information contained by the relation labels. We show an ablation study for this in Table 4, where R-GCN denotes the original R-GCN [32] and R-GCN w/ R-Emb denotes our version that considers relation labels.

Note, as mentioned in previous sections, the dataset to train, valid and test these four pre-training tasks are extracted from the *FTWP* dataset. There exist unseen nodes (ingredients in recipe) in the validation and test sets of *FTWP*, it requires strong generalizability to get decent performance on these datasets.

From Table 4, we show the relation label representation significantly boosts the generalization performance on these datasets. Compared to AP and SP, where relation label information has significant effect, both models perform near perfectly on the DGI task. This suggests the corruption function we consider in this work is somewhat simple, we leave this for future exploration.

Table 4: Test performance of models on all pre-training tasks.

Task	Graph Type	R-GCN	R-GCN w/ R-Emb
		Accuracy	
AP	full	0.472	**0.891**
	seen	0.631	**0.873**
SP	full	0.419	**0.926**
	seen	0.612	**0.971**
DGI	full	0.999	**1.000**
	seen	**1.000**	**1.000**

D Additional Results and Discussions

D.1 Training Curves

We report the training curves of all our mentioned experiment settings. Figure 11 shows the GATA's training curves. Figure 12 shows the training curves of the three text-based baseline (Tr-DQN, Tr-DRQN, Tr-DRQN+). Figure 13 shows the training curve of GATA-GTF (no graph updater, the action scorer takes ground-truth graphs as input) and GATA-GTP (graph updater is trained using ground-truth graphs *from the* FTWP *dataset*, the trained graph updater maintains a discrete belief graph throughout the RL training).

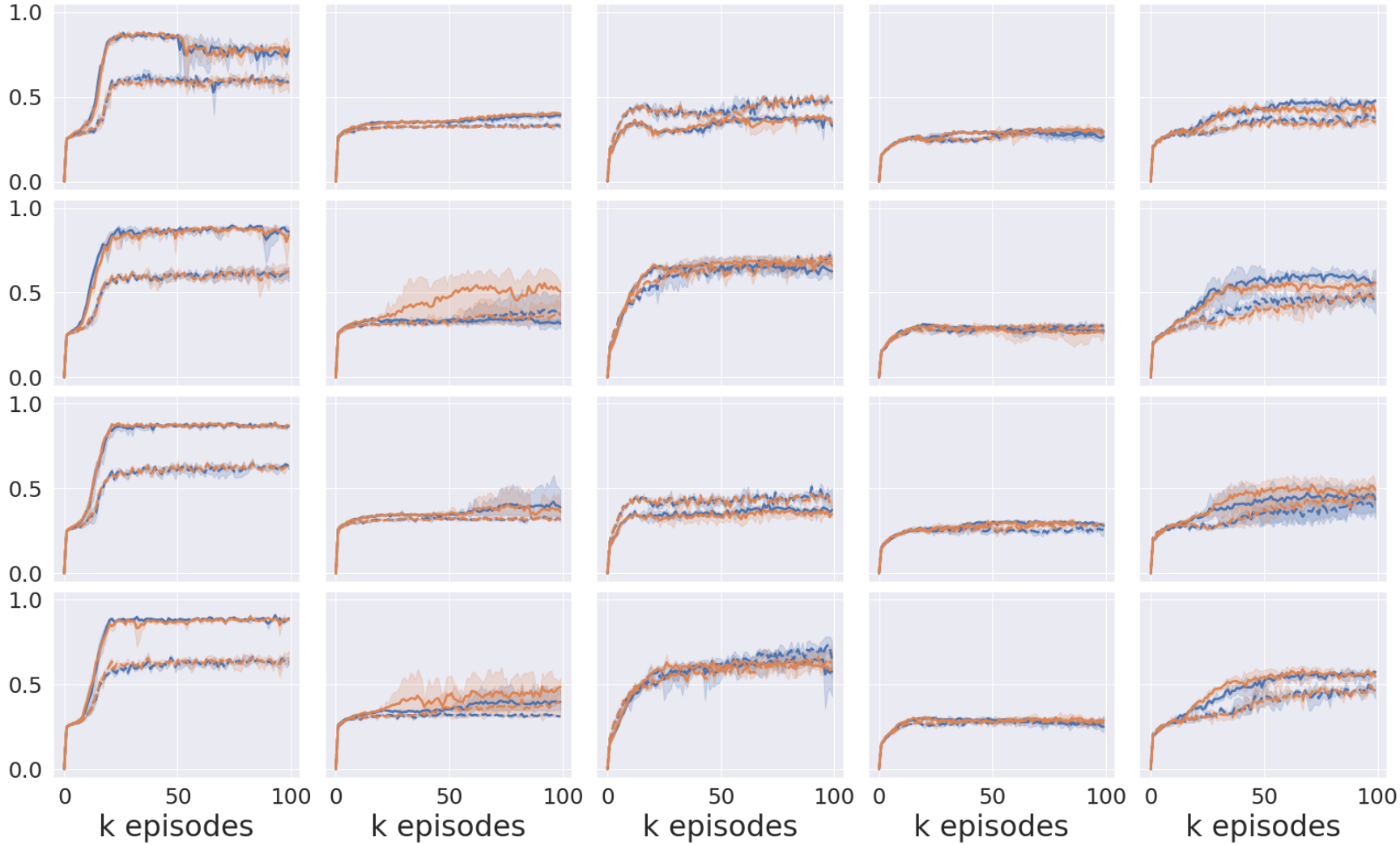


Figure 11: GATA's training curves (averaged over 3 seeds, band represents standard deviation). Columns are difficulty levels 1/2/3/4/5. The upper two rows are GATA using belief graphs generated by the graph updater pre-trained with observation generation task; The lower two rows are GATA using belief graphs generated by the graph updater pre-trained with contrastive observation classification task. In the 4 rows, the presence of text observation are False/True/False/True. In the figure, blue lines indicate the graph encoder in action selector is randomly initialized; orange lines indicate the graph encoder in action selector is initialized by the pre-trained observation generation and contrastive observation classification tasks. Solid lines indicate 20 training games, dashed lines indicate 100 training games.

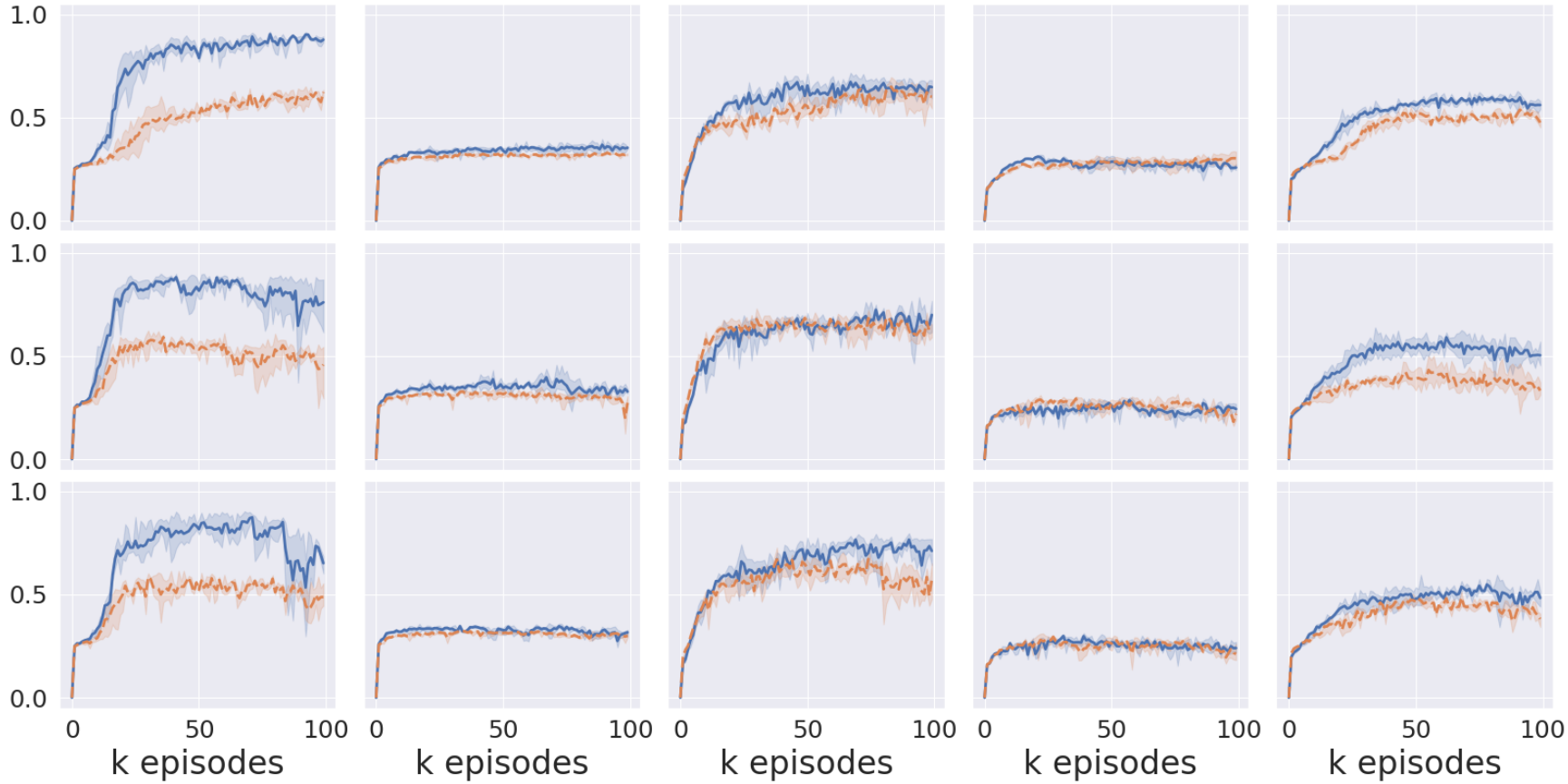


Figure 12: The text-based baseline agents' training curves (averaged over 3 seeds, band represents standard deviation). Columns are difficulty levels 1/2/3/4/5, rows are Tr-DQN, Tr-DRQN and Tr-DRQN+, respectively. All of the three agents take text observation O_t as input. In the figure, blue solid lines indicate the training set with 20 games; orange dashed lines indicate the training set with 100 games.

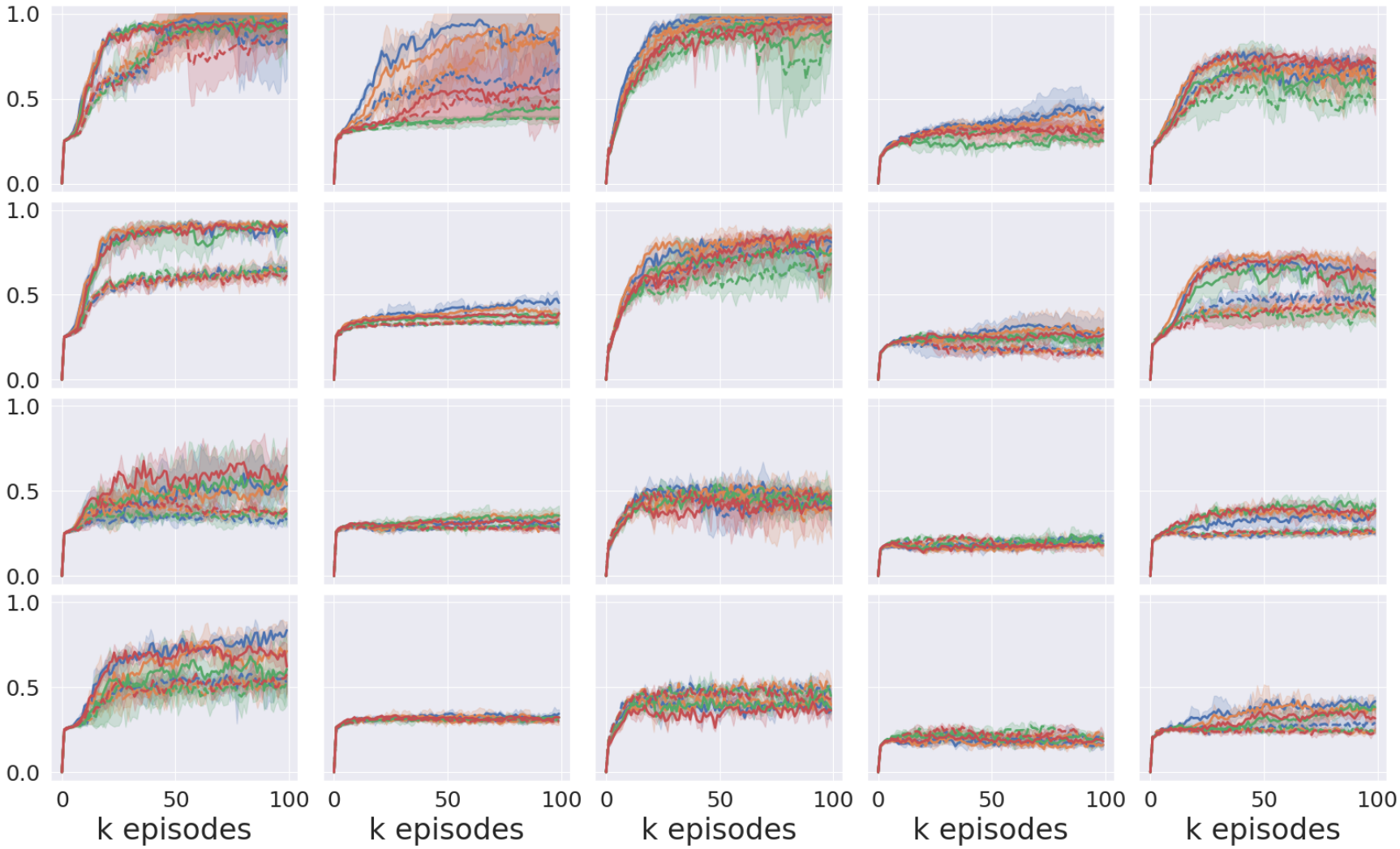


Figure 13: GATA-GTP and GATA-GTF's training curves (averaged over 3 seeds, band represents standard deviation). Columns are difficulty levels 1/2/3/4/5. The upper two rows are GATA-GTF when text observation is absent and present as input; the lower two rows are GATA-GTP when text observation is absent and present as input. In the figure, blue/orange/green indicate the agent's graph encoder is initialized with AP/SP/DGI pre-training tasks. Red lines indicate the graph encoder is randomly initialized. Solid lines indicate 20 training games, dashed lines indicate 100 training games.

D.2 Training Scores

In Table 5 we provide all agents' max training scores, each score is averaged over 3 random seeds. All scores are normalized. Note as described in Section 3.3, we use ground-truth KGs to train the action selector, $\mathcal{G}^{\text{belief}}$ is only used during evaluation.

Table 5: Agents' **Max** performance on **Training** games, averaged over 3 random seeds. In this table, ♠, ♦represent O_t and $\mathcal{G}_t^{\text{full}}$, respectively. ♣represents discrete belief graph generated by GATA-GTP (trained with ground-truth graphs of *FTWP*). ⋆and ∞indicate continuous belief graph generated by GATA, pre-trained with observation generation (OG) task and contrastive observation classification (COC) task, respectively. Light blue shadings represent numbers that are greater than or equal to Tr-DQN; light yellow shading represent number that are greater than or equal to all of Tr-DQN, Tr-DRQN and Tr-DRQN+.

		20 Training Games						100 Training Games						Avg.
Difficulty Level		1	2	3	4	5	% ↑	1	2	3	4	5	% ↑	% ↑
Input	Agent	Text-based Baselines												
♠	Tr-DQN	90.8	36.9	69.3	31.5	61.2	—	63.4	33.2	66.1	31.9	55.2	—	—
♠	Tr-DRQN	88.8	41.7	76.6	29.6	60.7	+2.9	60.8	33.7	71.7	30.6	44.9	-3.4	-0.2
♠	Tr-DQN+	89.1	35.6	78.0	30.9	58.1	+0.0	61.1	32.8	70.0	30.0	50.3	-2.8	-1.4
	Pre-training	GATA												
⋆	N/A	87.9	40.4	40.1	30.8	50.1	-11.2	65.1	34.6	51.2	32.0	41.8	-7.9	-9.6
⋆	OG	88.8	40.8	40.1	32.1	48.2	-10.6	63.8	33.9	51.9	32.6	39.3	-9.1	-9.8
⋆♠	N/A	90.2	35.4	69.0	32.0	62.8	-0.2	63.9	41.2	72.2	32.2	50.8	+5.4	+2.6
⋆♠	OG	90.0	57.1	70.6	31.7	57.9	+10.2	64.2	38.9	72.5	32.4	50.1	+4.1	+7.1
∞	N/A	89.0	43.1	41.2	31.8	48.7	-9.0	65.8	33.4	51.5	29.1	44.0	-9.4	-9.2
∞	COC	89.6	41.0	39.2	31.9	54.4	-8.7	65.8	33.4	48.6	31.2	47.2	-7.8	-8.2
∞♠	N/A	90.9	41.4	66.1	31.3	58.8	+0.6	67.1	33.1	73.6	29.9	51.2	+0.7	+0.6
∞♠	COC	90.2	50.8	66.4	31.9	59.3	+6.2	67.4	41.5	66.6	31.4	50.8	+4.5	+5.4
		GATA-GTP												
♣	N/A	73.3	34.5	50.5	21.7	43.5	-22.6	49.3	31.1	54.3	25.1	28.8	-23.1	-22.8
♣	AP	68.4	34.8	61.3	23.8	43.1	-19.2	40.9	31.3	55.4	24.8	28.8	-25.7	-22.4
♣	SP	62.7	38.1	57.5	23.5	44.1	-19.6	50.2	30.8	55.0	23.7	28.0	-24.0	-21.8
♣	DGI	64.9	37.0	55.6	25.7	47.4	-17.8	43.4	31.8	58.3	25.3	30.2	-22.7	-20.3
♣♠	N/A	77.5	33.9	45.6	26.3	40.2	-21.6	59.5	32.3	55.4	29.1	27.6	-16.8	-19.2
♣♠	AP	87.5	35.8	50.4	22.3	45.4	-17.8	61.3	32.2	56.3	25.3	33.0	-16.4	-17.1
♣♠	SP	80.0	35.5	50.2	23.5	44.0	-19.4	57.3	32.1	58.3	27.1	29.2	-17.4	-18.4
♣♠	DGI	70.3	33.9	51.4	26.3	42.1	-20.9	57.7	32.7	55.6	28.8	29.8	-16.4	-18.6
		GATA-GTF												
♦	N/A	98.6	58.4	95.6	36.1	80.9	+30.3	96.0	53.4	97.9	36.0	76.4	+42.3	+36.3
♦	AP	98.7	97.5	98.3	48.1	79.3	+59.4	97.1	74.7	98.3	44.5	75.9	+60.8	+60.1
♦	SP	100.0	96.9	98.3	44.9	76.6	+56.5	98.6	90.5	99.0	38.9	73.4	+66.6	+61.5
♦	DGI	96.9	45.4	95.3	28.7	72.6	+15.4	98.2	39.1	90.1	33.0	62.4	+25.1	+20.2
♦♠	N/A	91.7	55.9	80.9	33.6	63.2	+15.8	73.5	48.1	67.7	31.8	56.7	+13.1	+14.5
♦♠	AP	87.9	62.4	78.8	32.4	62.8	+17.0	76.8	54.0	73.7	34.1	55.6	+20.6	+18.8
♦♠	SP	90.7	55.8	83.8	30.7	64.2	+14.9	60.4	40.1	67.4	31.1	51.5	+1.8	+8.3
♦♠	DGI	88.1	38.1	73.0	32.5	62.5	+2.2	66.4	35.5	59.1	30.4	49.6	-2.8	-0.3

D.3 Test Results

In Table 6 we provide all our agent variants and the text-based baselines' test scores. We report agents' test score corresponding to their best validation scores.

Table 6: Agents' performance on **test** games, model selected using best validation performance. **Boldface** and underline represent the highest and second highest values in a setting (excluding GATA-GTF which has access to the ground-truth graphs of the RL games). In this tabel, ♠, ♦represent O_t and $\mathcal{G}_t^{\text{full}}$, respectively. ♣represents discrete belief graph generated by GATA-GTP (pre-trained with ground-truth graphs of *FTWP*). ⋆and ∞indicate continuous belief graph generated by GATA, pre-trained with observation generation (OG) task and contrastive observation classification (COC) task, respectively. Light blue shadings represent numbers that are greater than or equal to Tr-DQN; light yellow shading represent number that are greater than or equal to all of Tr-DQN, Tr-DRQN and Tr-DRQN+.

		20 Training Games						100 Training Games						Avg.
Difficulty Level		1	2	3	4	5	% ↑	1	2	3	4	5	% ↑	% ↑
Input	Agent	Text-based Baselines												
♠	Tr-DQN	66.2	26.0	16.7	18.2	**27.9**	—	62.5	32.0	38.3	17.7	34.6	—	—
♠	Tr-DRQN	62.5	32.0	28.3	12.7	26.5	+10.3	58.8	31.0	36.7	21.4	27.4	-2.6	+3.9
♠	Tr-DRQN+	65.0	30.0	35.0	11.8	18.3	+10.7	58.8	33.0	33.3	19.5	30.6	-3.4	+3.6
	Pre-training	GATA												
⋆	N/A	70.0	20.0	20.0	18.6	26.3	-0.2	62.5	32.0	46.7	**27.7**	35.4	+16.1	+8.0
⋆	OG	66.2	28.0	21.7	15.9	24.3	+2.4	66.2	34.0	40.0	21.4	34.0	+7.2	+4.8
⋆♠	N/A	66.2	34.0	30.0	12.7	24.3	+13.5	66.2	**38.0**	36.7	27.3	36.1	+15.8	+14.6
⋆♠	OG	66.2	**48.0**	26.7	15.5	26.3	+24.8	66.2	36.0	**58.3**	14.1	**45.0**	+16.1	+20.4
∞	N/A	73.8	42.0	26.7	**20.9**	24.5	+27.1	62.5	30.0	51.7	23.6	36.0	+13.2	+20.2
∞	COC	66.2	29.0	30.0	18.2	27.7	+18.1	66.2	34.0	41.7	19.1	40.3	+9.1	+13.6
∞♠	N/A	68.8	33.0	41.7	17.7	27.0	+34.9	62.5	33.0	46.7	25.9	33.4	+13.6	+24.2
∞♠	COC	66.2	44.0	16.7	20.0	21.7	+11.4	**70.0**	34.0	45.0	12.3	36.2	+2.0	+6.7
		GATA-GTP												
♣	N/A	56.2	23.0	41.7	11.4	22.1	+13.0	45.0	32.0	30.0	10.5	17.4	-28.0	-7.5
♣	AP	50.0	20.0	25.0	9.5	24.3	-11.7	45.0	31.0	50.0	15.9	24.4	-8.0	-9.9
♣	SP	45.0	25.0	38.3	11.8	22.6	+7.9	42.5	32.0	50.0	11.4	22.5	-14.4	-3.3
♣	DGI	56.2	26.0	40.0	17.3	17.7	+16.6	37.5	31.0	45.0	13.6	18.7	-18.9	-1.2
♣♠	N/A	73.8	31.0	28.3	8.2	22.5	+5.2	62.5	29.0	38.3	13.2	19.8	-15.5	-5.2
♣♠	AP	62.5	32.0	**46.7**	12.3	21.1	+28.1	58.8	30.0	40.0	10.9	29.2	-12.4	+7.9
♣♠	SP	65.0	32.0	41.7	12.3	23.5	+24.6	62.5	32.0	51.7	21.8	23.5	+5.2	+14.9
♣♠	DGI	**75.0**	27.0	33.3	17.3	24.3	+19.7	62.5	31.0	46.7	19.5	24.7	+0.1	+9.9
		GATA-GTF												
♦	N/A	83.8	53.0	33.3	23.6	24.8	+49.7	100.0	90.0	68.3	37.3	52.7	+96.5	+73.1
♦	AP	85.0	39.0	26.7	26.4	27.5	+36.4	92.5	88.0	63.3	53.6	51.6	+108.0	+72.2
♦	SP	48.7	61.0	46.7	23.6	28.9	+64.2	95.0	95.0	70.0	37.3	52.8	+99.0	+81.6
♦	DGI	85.0	27.0	31.7	14.1	22.1	+15.7	100.0	40.0	70.0	31.8	50.6	+58.7	+37.2
♦♠	N/A	92.5	39.0	30.0	15.9	23.6	+28.3	96.3	56.0	55.0	14.5	46.6	+37.9	+33.1
♦♠	AP	73.8	36.0	46.7	25.9	23.9	+51.5	85.0	42.0	68.3	36.4	47.5	+57.7	+54.6
♦♠	SP	62.5	24.0	36.7	14.5	28.6	+17.7	60.0	43.0	46.7	25.0	47.9	+26.4	+21.1
♦♠	DGI	81.2	30.0	25.0	16.8	30.7	+18.0	73.8	39.0	48.3	15.0	40.6	+13.6	+15.8

D.4 Other Remarks

Pre-training graph encoder helps. In Table 5 and Table 6, we also show GATA, GATA-GTP and GATA-GTF's training and test scores when their action scorer's graph encoder are initialized with pre-trained parameters as introduced in Section 3.2 (for GATA) and Appendix C.3 (for GATA-GTP and GATA-GTF). We observe in most settings, pre-trained graph encoders produce better training and test results compared to their randomly initialized counterparts. This is particularly obvious in GATA-GTP and GATA-GTF, where graphs are discrete. For instance, from Table 6 we can see that only with text observation as additional input (♣♠), and when graph encoder are initialized with AP/SP/DGI, the GATA-GTP agent can outperform the text-based baselines on test game sets.

Fine-tuning graph encoder helps. For all experiment settings where the graph encoder in action scorer is initialized with pre-trained parameters (OG/COC for GATA, AP/SP/DGI for GATA-GTP), we also compare between freezing vs. fine-tuning the graph encoder in RL training. By freezing the graph encoders, we can effectively reduce the number of parameters to be optimized with RL signal. However, we see consistent trends that fine-tuning the graph encoders can always provide better training and testing performance in both GATA and GATA-GTP.

Text input helps more when graphs are imperfect. We observe clear trends that for GATA-GTF, using text together with graph as input (to the action selector) does not provide obvious performance increase. Instead, GATA-GTF often shows better performance when text observation input is disabled. This observation is coherent with the intuition of using text observations as additional input. When the input graph to the action selector is imperfect (e.g., belief graph maintained by GATA or GATA-GTP itself), the text observation provides more accurate information to help the agent to recover from errors. On the other hand, GATA-GTF uses the ground-truth full graph (which is even accurate than text) as input to the action selector, the text observation might confuse the agent by providing redundant information with more uncertainty.

Learning across difficulty levels. We have a special set of RL games — level 5 — which is a mixture of the other four difficulty levels. We use this set to evaluate an agent's generalizability on both dimensions of game configurations and difficulty levels. From Table 5, we observe that almost all agents (including baseline agents) benefit from a larger training set, i.e., achieve better test results when train on 100 level 5 games than 20 of them. Results show GATA has a more significant performance boost from larger training set. We notice that all GATA-GTP variants perform worse than text-based baselines on level 5 games, whereas GATA outperforms text-based baselines when training on 100 games. This may suggest the continuous belief graphs can better help GATA to adapt to games across difficulty levels, whereas its discrete counterpart may struggle more. For example, both games in level 1 and 2 have only single location, while level 3 and 4 games have multiple locations. GATA-GTP might thus get confused since sometimes the direction relations (e.g., `west_of`) are unused. In contrast, GATA, equipped with continuous graphs, may learn such scenario easier.

D.5 Probing Task and Belief Graph Visualization

In this section, we investigate whether generated belief graphs contain any useful information about the game dynamics. We first design a probing task to check if $\mathcal{G}$ encodes the existing relations between two nodes. Next, we visualize a few slices of the adjacency tensor associated to $\mathcal{G}$.

D.5.1 Probing Task

We frame the probing task as a multi-label classification of the relations between a pair of nodes. Concretely, given two nodes i, j, and the vector $\mathcal{G}_{i,j} \in [-1, 1]^{\mathcal{R}}$ (in which $\mathcal{R}$ denotes the number of relations) extracted from the belief graph $\mathcal{G}$ corresponding to the nodes i and j, the task is to learn a function f such that it minimizes the following binary cross-entropy loss:

$$\mathcal{L}_{\text{BCE}}\left(f(\mathcal{G}_{i,j}, h_i, h_j), Y_{i,j}\right), \tag{15}$$

where h_i, h_j are the embeddings for nodes i and j, $Y_{i,j} \in \{0, 1\}^{\mathcal{R}}$ is a binary vector representing the presence of each relation between the nodes (there are R different relations). Following Alain and Bengio [2], we use a linear function as f, since we assume the useful information should be easily accessible from $\mathcal{G}$.

Table 7: Probing task results showing that belief graphs obtained from OG and COC do contain information about the game dynamics, i.e. node relationships.

	Exact Match						F_1 score					
	Train			Test			Train			Test		
Model	+	−	Avg	+	−	Avg	+	−	Avg	+	−	Avg
Random	0.00	0.99	0.49	0.00	0.99	0.49	0.00	0.99	0.49	0.00	0.99	0.49
Ground-truth	0.98	0.96	0.97	0.97	0.96	0.97	0.98	0.96	0.97	0.98	0.96	0.97
Tr-DRQN	0.61	0.84	0.73	0.61	0.83	0.72	0.61	0.84	0.73	0.61	0.83	0.72
GATA (OG)	0.69	0.86	**0.78**	0.70	0.86	**0.78**	0.71	0.85	**0.78**	0.72	0.86	**0.79**
GATA (COC)	0.65	0.86	0.75	0.65	0.84	0.75	0.67	0.85	0.76	0.67	0.84	0.75

We collect a dataset for this probing task by following the walkthroughs of 120 games. At every game step, we collect a tuple $(\mathcal{G}, \mathcal{G}^{\text{seen}})$ (see Appendix C.2 for the definition of $\mathcal{G}^{\text{seen}}$). We used tuples from 100 games as training data and the remaining for evaluation.

From each tuple in the dataset, we extract several node pairs (i, j) and their corresponding $Y_{i,j}$ from $\mathcal{G}^{\text{seen}}$ (positive examples, denoted as "+"). To make sure a model can only achieve good performance on this probing task by using the belief graph $\mathcal{G}$, without overfitting by memorising node-relation pairs (e.g., the unique relation between player and kitchen is `at`), we augment the dataset by adding plausible node pairs (i.e., $Y_{i,j} = \vec{0}^{\mathcal{R}}$) but that have no relation according to the current $\mathcal{G}$ (negative examples, denoted as "−"). For instance, if at a certain game step the player is in the bedroom, the relation between the player and kitchen should be empty ($\vec{0}^{\mathcal{R}}$). We expect $\mathcal{G}$ to have captured that information.

We use two metrics to evaluate the performance on this probing task:
- **Exact match** represents the percentage of predictions that have all their labels classified correctly, i.e., when $f(\mathcal{G}_{i,j}, h_i, h_j) = Y^n_{i,j}$.
- **F_1 score** which is the harmonic mean between precision and recall. We report the macro-averaging of F_1 over all the predictions.

To better understand the probe's behaviors on each settings, we also report their training and test performance on the positive samples (+) and negative samples (−) separately.

From Table 7, we observe that belief graphs $\mathcal{G}$ generated by models pre-trained with either OG or COC do contain useful information about the relations between a pair of nodes. We first compare against a random baseline where each $\mathcal{G}$ is randomly sampled from $\mathcal{N}(0, 1)$ and kept fixed throughout the probing task. We observe the linear probe fails to perform well on the training set (and as a result also fails to generalize on test set). Interestingly, with random belief graphs provided, the probe somehow overfits on negative samples and always outputs zeros all the time. In both training and testing phases, it produces zero performance on positive examples. This baseline suggests the validity of our probing task design — there is no way to correctly predict the relations without having the information encoded in the belief graph $\mathcal{G}$.

Next, we report the performance of using ground-truth graphs ($\mathcal{G}^{\text{seen}}$) as input to f. We observe the linear model can perform decently on training data, and can generalize from training to testing data — on both sets, the linear probe achieves near-perfect performance. This also verifies the probing task by showing that given the ground-truth knowledge, the linear probe is able to solve the task easily.

Given the two extreme cases as upper bound and lower bound, we investigate the belief graphs $\mathcal{G}$ generated by GATA, pre-trained with either of the two self-supervised methods, OG and COC (proposed in Section 3.2). From Table 7, we can see $\mathcal{G}$ generated by both OG and COC methods help similarly in the relation prediction task, both provide more than 75% of testing exact match scores.

In Section 4, we show that GATA outperforms a set of baseline systems, including Tr-DRQN (as described in Section 4.1), an agent with recurrent components. To further investigate if the belief graphs generated by GATA can better facilitate the linear probe in this relation prediction task, we provide an additional setting. We modify the Tr-DRQN agent by replacing its action scorer by a text generator (the same decoder used in OG training), and train this model with the same data and objective as OG. After pre-training, we obtain a set of probing data by collecting the recurrent hidden states produced by this agent given the same probing game walkthroughs. Since these recurrent

hidden states are computed from the same amount of information as GATA's belief graphs, they could theoretically contain the same information as $\mathcal{G}$. However, from Table 7, we see that the scores of Tr-DRQN are consistently lower than GATA's score. This is coherent with our findings in the RL experiments (Section 4), except the gap between GATA and Tr-DRQN is less significant in the relation prediction task setting.

While being able to perform the classification correctly in a large portion of examples, we observe a clear performance gap comparing GATA's belief graphs with ground-truth graphs. The cause of the performance gap can be twofold. First, compared to ground-truth graphs that accurately represent game states without information loss, $\mathcal{G}$ (iteratively generated by a neural network across game steps) can inevitably suffer from information loss. Second, the information encoded in $\mathcal{G}$ might not be easily extracted by a linear probe (compared to ground-truth). Both aspects suggest potential future directions to improve the belief graph generation module.

We optimize all probing models for 10 epochs with Adam optimizer, using the default hyperparameters and a learning rate of 0.0001. Note in all the probing models, only parameters of the linear layer f are trainable, everything else (including node embeddings) are kept fixed.

D.5.2 Belief Graph Visualization

In Figure 14, we show a slice of the ground-truth adjacency tensor representing the `is` relation. To give context, that tensor has been extracted at the end of a game with a recipe requesting a fried diced red apple, a roasted sliced red hot pepper, and a fried sliced yellow potato. Correspondingly, for the same game and same time step, Figure 15 shows the same adjacency tensor's slice for the belief graphs $\mathcal{G}$ generated by GATA pre-trained on observation generation (OG) and contrastive observation classification (COC) tasks.

For visualization, we found that subtracting the mean adjacency tensor, computed across all games and steps, helps by removing information about the marginal distribution of the observations (e.g., underlying grammar or other common features needed for the self-supervised tasks). Those "cleaner" graphs are shown in Figure 16. One must keep in mind that there is no training signal to force the belief graphs to align with any ground-truth graphs since the belief graph generators are trained with pure self-supervised methods.

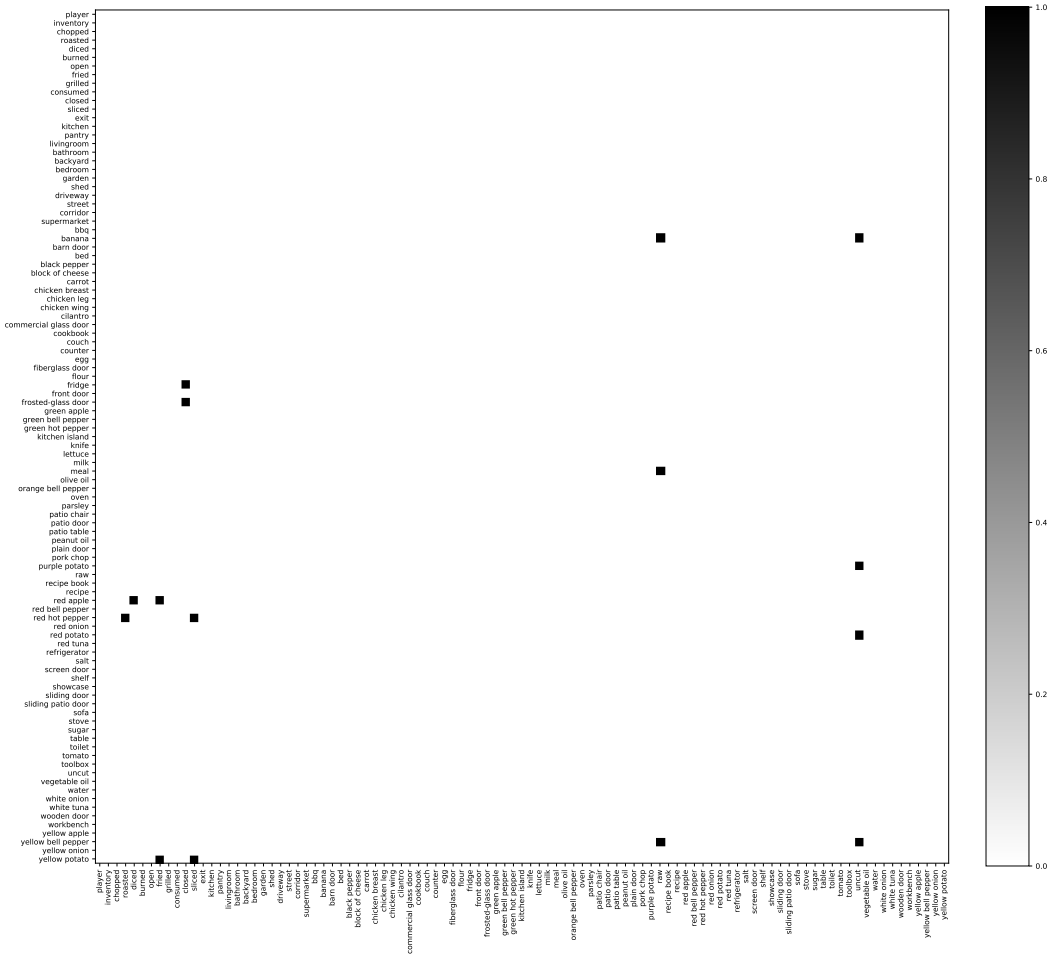


Figure 14: Slice of a ground-truth adjacency tensor representing the `is` relation.

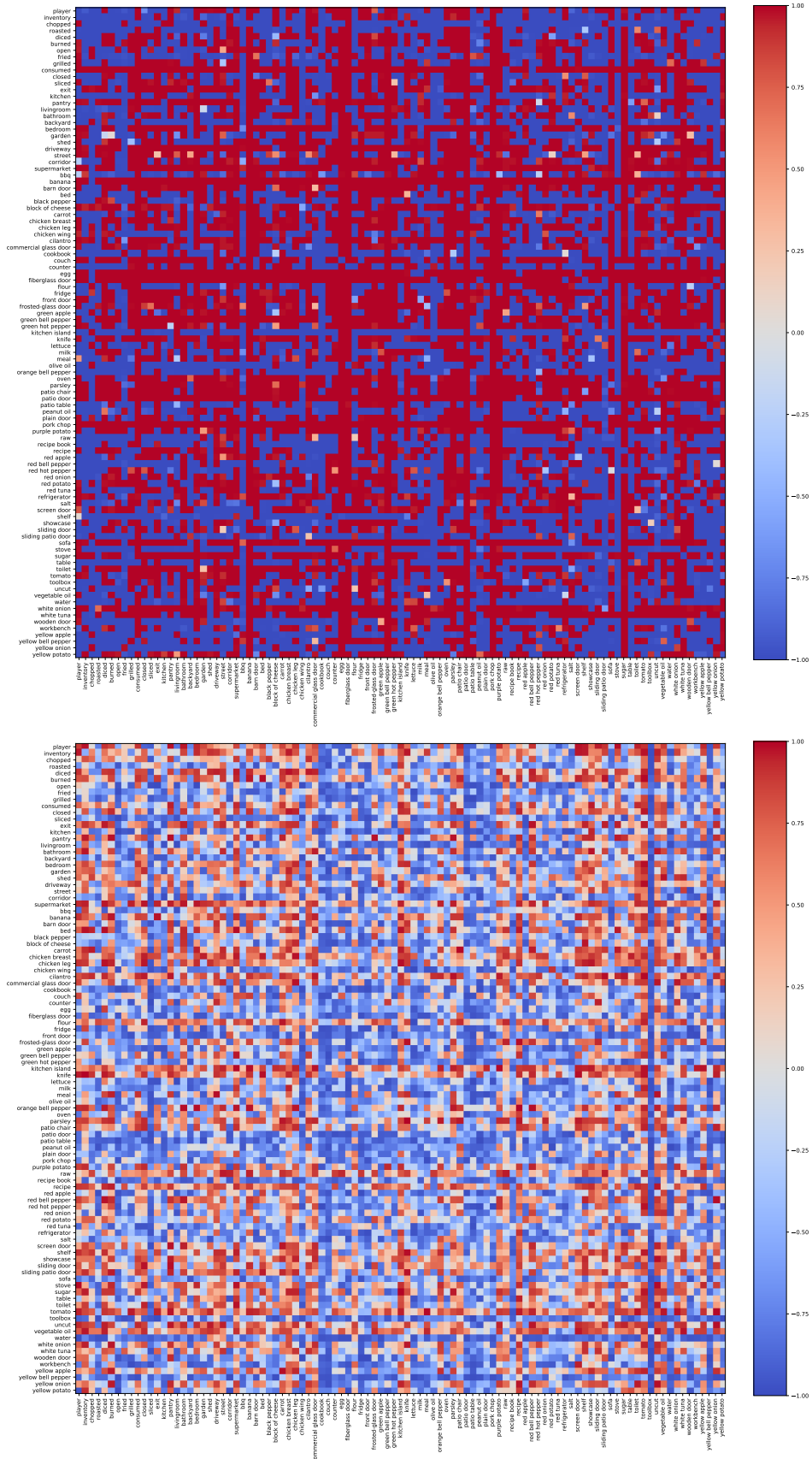


Figure 15: Adjacency tensor's slices for $\mathcal{G}$ generated by GATA, pre-trained with OG task **(top)** and COC task **(bottom)**.

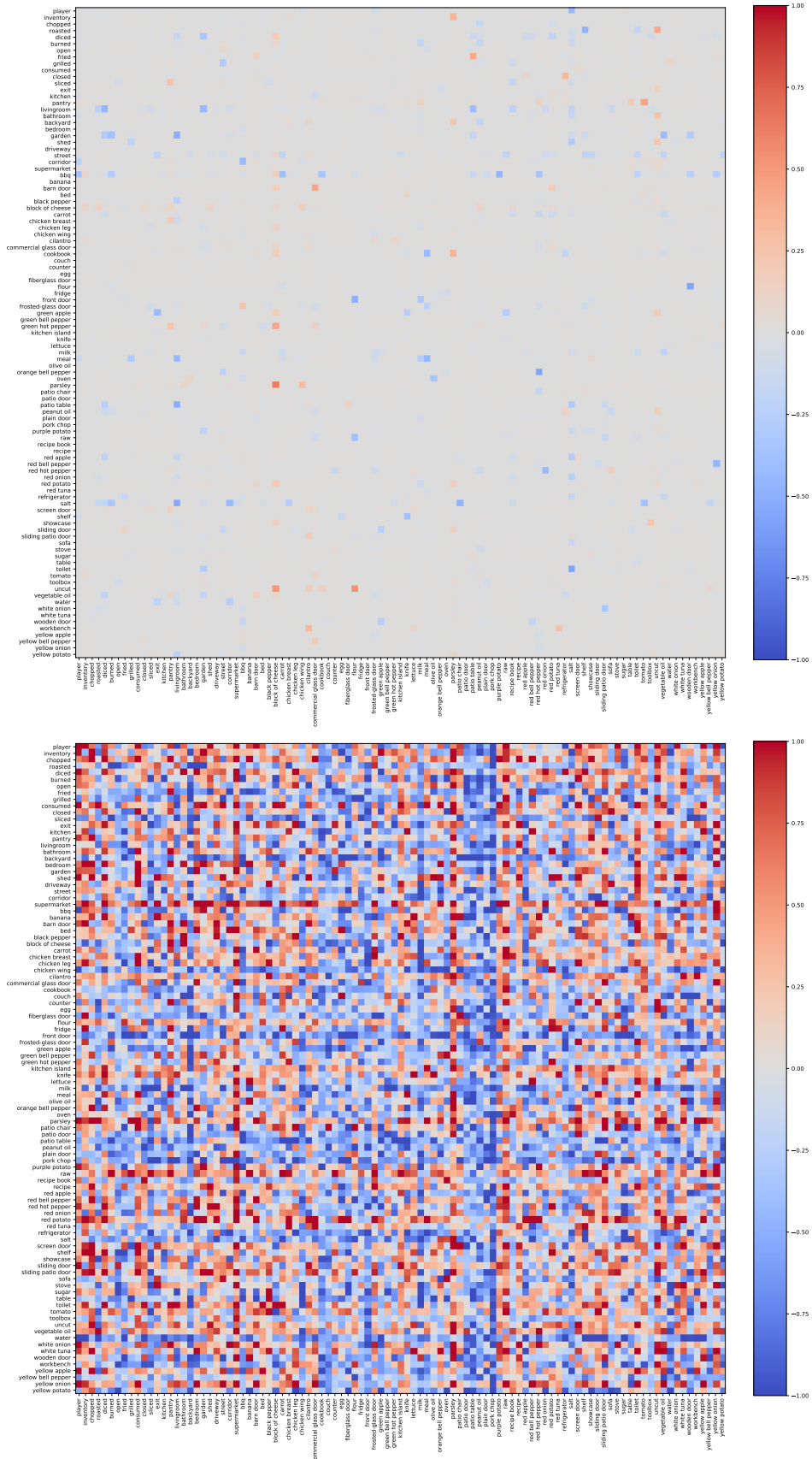


Figure 16: Adjacency tensor's slices after subtracting the mean for $\mathcal{G}$ generated by GATA, pre-trained with OG task **(top)** and COC task **(bottom)**.

Table 8: Selected example of test set of the observation generation task. On this specific data point, The model gets an F_1 score of 0.64. We use same colors to shade the information overlap between ground-truth observation with the model prediction.

Ground-Truth
-= kitchen = - of every kitchen you could have sauntered into , you had to saunter into a normal one . you can make out a fridge . the fridge contains a raw chicken wing , a raw chicken leg and a block of cheese . you can make out an oven . you hear a noise behind you and spin around , but you ca n't see anything other than a table . wow , is n't textworld just the best ? you see a cookbook on the table . hmmm ... what else , what else ? you make out a counter . now why would someone leave that there ? the counter is vast . on the counter you make out a raw purple potato and a knife . you can make out a stove . the stove is conventional . unfortunately , there is n't a thing on it . hopefully this does n't make you too upset . there is an open frosted - glass door leading east . there is an exit to the north . do n't worry , there is no door . you need an exit without a door ? you should try going west . there is a white onion on the floor .
Prediction
-= kitchen = - you 're now in a kitchen . you begin looking for stuff . you can make out a fridge . the fridge contains a raw chicken leg . you can make out a closed oven . you can make out a table . the table is massive . on the table you see a knife . you see a counter . the counter is vast . on the counter you can make out a cookbook . you can make out a stove . the stove is conventional . but the thing has n't got anything on it . you can make out a stove . the stove is conventional . but the thing has n't got anything on it . there is an open frosted - glass door leading east . there is an exit to the east .

D.6 GATA's Performance on Graph Updater Pre-training Tasks

In this subsection, we report the test performance of the two graph updater pre-training tasks for GATA.

In the observation generation (OG) task, our model achieves a test loss of 22.28 (when using ground-truth history tokens as input to the decoder), and a test F_1 score of 0.68 (when using the previously generated token as input to the decoder).

In the contrastive observation classification (COC) task, our model achieves a test loss of 0.06, and a test accuracy of 0.97.

In Table 8, we show a selected test example from the observation generation task. In the example, the prediction has fair amount of overlap (semantically, not necessarily word by word) with the ground-truth. This suggests the belief graph $\mathcal{G}_t^{\text{belief}}$ generated by GATA's updater model can to some extent capture and encode the state information of the environment — since the model is able to reconstruct O_t using $\mathcal{G}_t^{\text{belief}}$.

D.7 Future Directions

As mentioned in Appendix D.5, the belief graphs generated by GATA lack of interpretability because the training is not supervised by any ground-truth graph. Technically, they are recurrent hidden states that encode the game state, we only (weakly) ground these real-valued graphs by providing node and relation vocabularies (word embeddings) for the message passing in R-GCN.

Therefore, there can be two potential directions deriving from the current approach. First, it would be interesting to investigate regularization methods and auxiliary tasks that can make the belief graph sparser (without relying on ground-truth graphs to train). A sparser belief graph may increase GATA's interpretability, however, it does not guarantee to produce better performance on playing text-based games (which is what we care more about).

Second, it would also be interesting to see how GATA can be adapted to environments where the node and relation names are unknown. This will presumably make the learned belief graphs even far away from interpretable, but at the same time it will further relax GATA from the need of requiring any prior knowledge about the environments. We believe this is an essential property for an agent that is generalizabile to out-of-distribution environments. For instance, without the need of a pre-defined node and relation vocabularies, we can expand GATA to the setting where training on the cooking games, and testing on games from another genre, or even text-based games designed for humans [14].

Algorithm 1 Training Strategy for GATA Action Selector

1: **Input:** games $\mathcal{X}$, replay buffer B, update frequency F, patience P, tolerance τ, evaluation frequency E.
2: Initialize counters $k \leftarrow 1, p \leftarrow 0$, best validation score $V \leftarrow 0$, transition cache C, policy π, checkpoint Π.
3: **for** $e \leftarrow 1$ **to** NB_EPISODES **do**
4: Sample a game $x \in \mathcal{X}$, reset C.
5: **for** $i \leftarrow 1$ **to** NB_STEPS **do**
6: play game, push transition into C, $k \leftarrow k + 1$
7: **if** $k\%F = 0$ **then** sample batch from B, Update(π)
8: **if** done **then** break
9: **end for**
10: **if** average score in $C > \tau \cdot$ average score in B **then**
11: **for all** item in C **do** push item into B
12: **end if**
13: **if** $e\%E \neq 0$ **then** continue
14: $v \leftarrow$ Evaluate(π)
15: **if** $v >= V$ **then** $\Pi \leftarrow \pi$, $p \leftarrow 0$, continue
16: **if** $p > P$ **then** $\pi \leftarrow \Pi$, $p \leftarrow 0$
17: **else** $p \leftarrow p + 1$
18: **end for**

E Implementation Details

In Appendix A, we have introduced hyperparameters regarding model structures. In this section, we provide hyperparameters we used in training and optimizing.

In all experiments, we use *Rectified Adam* [25] as the step rule for optimization. The learning rate is set to 0.001. We clip gradient norm of 5.

E.1 Graph Updater

To pre-train the recurrent graph updater in GATA, we utilize the backpropagation through time (BPTT) algorithm. Specifically, we unfold the recurrent graph updater and update every 5 game steps. We freeze the parameters in graph updater after its own training process. Although it can theoretically be trained on-the-fly together with the action selector (with reward signal), we find the standalone training strategy more effective and efficient.

E.2 Action Selector

The overall training procedure of GATA's action selector is shown in Algorithm 1.

We report two strategies that we empirically find effective in DQN training. First, we discard the underachieving trajectories without pushing them into the replay buffer (lines 10–12). Specifically, we only push a new trajectory that has an average reward greater than $\tau \in \mathbb{R}_0^+$ times the average reward for all transitions in the replay buffer. We use $\tau = 0.1$, since it keeps around some weaker but acceptable trajectories and does not limit exploration too severely. Second, we keep track of the best performing policy Π on the validation games. During training, when GATA stops improving on validation games, we load Π back to the training policy π and resume training. After training, we report the performance of Π on test games. Note these two strategies are not designed specifically for GATA; rather, we find them effective in DQN training in general.

We use a prioritized replay buffer with memory size of 500,000, and a priority fraction of 0.6. We use ϵ-greedy, where the value of ϵ anneals from 1.0 to 0.1 within 20,000 episodes. We start updating parameters after 100 episodes of playing. We update our network after every 50 game steps (update frequency F in Algorithm 1) [7]. During update, we use a mini-batch of size 64. We use a discount $\gamma = 0.9$. We update target network after every 500 episodes. For multi-step learning, we sample the multi-step return $n \sim \text{Uniform}[1, 3]$. We refer readers to Hessel et al. [18] for more information about different components of DQN training.

[7] 50 is the total steps performed within a batch. For instance, when batch size is 1, we update per 50 steps; whereas when batch size is 10, we update per 5 steps. Note the batch size here refers to the parallelization of the environment, rather than the batch size for backpropagation.

In our implementation of the Tr-DRQN and Tr-DRQN+ baselines, following Yuan et al. [50], we sample a sequence of transitions of length 8, use the first 4 transitions to estimate reasonable recurrent states and use the last for to update. For counting bonus, we use a $\gamma_c = 0.5$, the bonus is scaled by an coefficient $\lambda_c = 0.1$.

For all experiment settings, we train agents for 100,000 episodes (NB_EPISODES in Algorithm 1). For each game, we set maximum step of 50 (NB_STEPS in Algorithm 1). When an agent has used up all its moves, the game is forced to terminate. We evaluate them after every 1,000 episodes (evaluation frequency E in Algorithm 1). Patience P and tolerance τ in Algorithm 1 are 3 and 0.1, respectively. The agents are implemented using PyTorch [30].

E.3 Wall Clock Time

We report our experiments' wall clock time. We run all the experiments on single NVIDIA P40/P100 GPUs.

Table 9: Wall clock time for all experiments.

Setting/Component	Batch Size	Approximate Time
GATA		
Graph Updater - OG (Section 3.2)	48	2 days
Graph Updater - COC (Section 3.2)	64	2 days
Action Scorer (Section 3.3)	64 (backprop)	2 days
GATA-GTP		
Discrete Graph Updater (Appendix C.2)	128	2 day
Action Scorer (same as (Section 3.3))	64 (backprop)	2 days
GATA-GTF		
Action Scorer (same as (Section 3.3))	64 (backprop)	2 days
Discrete Graph Encoder Pre-training		
Action Prediction w/ full graph, for GATA-GTF (Appendix C.3)	256	2 days
Action Prediction w/ seen graph, for GATA-GTP (Appendix C.3)	256	2 days
State Prediction w/ full graph, for GATA-GTF (Appendix C.3)	48	5 days
State Prediction w/ seen graph, for GATA-GTP (Appendix C.3)	48	5 days
Deep Graph Infomax w/ full graph, for GATA-GTF (Appendix C.3)	256	1 day
Deep Graph Infomax w/ seen graph, for GATA-GTP (Appendix C.3)	256	1 day
Text-based Baselines		
Tr-DQN (Section 4.1)	64 (backprop)	2 days
Tr-DRQN (Section 4.1)	64 (backprop)	2 days
Tr-DRQN+ (Section 4.1)	64 (backprop)	2 days

F Details of the *FTWP* dataset

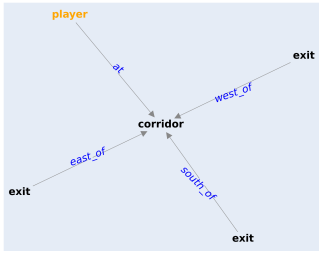
Previously, Trischler et al. [39] presented the *First TextWorld Problems (FTWP)* dataset, which consists of TextWorld games that follow a cooking theme across a wide range of difficulty levels. Although this dataset is analogous to what we use in this work, it has only 10 games per difficulty level. This is insufficient for reliable experiments on generalization, so we generate new game sets for our work. As mentioned in Section 3.2 and Appendix B, we use a set of transitions collected from the *FTWP* dataset. To ensure the fairness of using this dataset, we make sure there is no overlap between the *FTWP* and the games we use to train and evaluate our action selector.

Extracting Ground-truth Graphs from *FTWP* Dataset

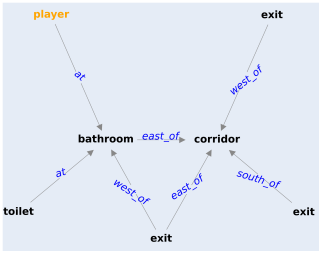
Under the hood, TextWorld relies on predicate logic to handle the game dynamics. Therefore, the underlying game state consists of a set of predicates, and logic rules (i.e. actions) can be applied to update them. TextWorld's API allows us to obtain such underlying state S_t at a given game step t for any games generated by the framework. We leverage S_t to extract both $\mathcal{G}_t^{\text{full}}$ and $\mathcal{G}_t^{\text{seen}}$.

In which, $\mathcal{G}_t^{\text{full}}$ is a discrete KG that contains the full information of the current state at game step t; $\mathcal{G}_t^{\text{seen}}$ is a discrete partial KG that contains information the agent has observed from the beginning until step t.

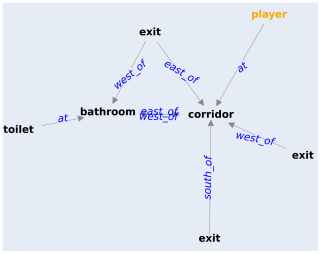
Figure 17 shows an example of consecutive $\mathcal{G}_t^{\text{seen}}$ as the agent explores the environment of a *FTWP* game. Figure 18 shows the $\mathcal{G}^{\text{full}}$ extracted from the same game.



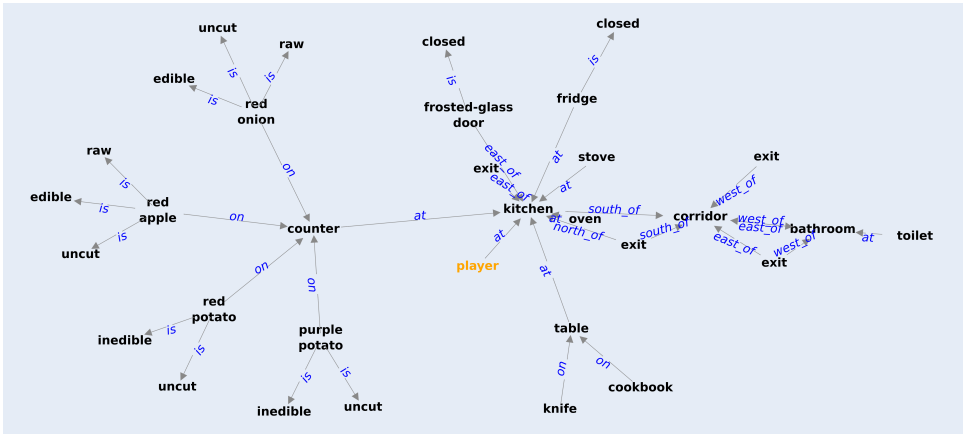
(a) $\mathcal{G}_0^{\text{seen}}$ after starting the game.



(b) $\mathcal{G}_1^{\text{seen}}$ after `go east`.



(c) $\mathcal{G}_2^{\text{seen}}$ after `go west`.



(d) $\mathcal{G}_3^{\text{seen}}$ after `go south` that leads to the kitchen which contains many objects.

Figure 17: A sequence of $\mathcal{G}^{\text{seen}}$ extracted after issuing three consecutive actions in a *FTWP* game.

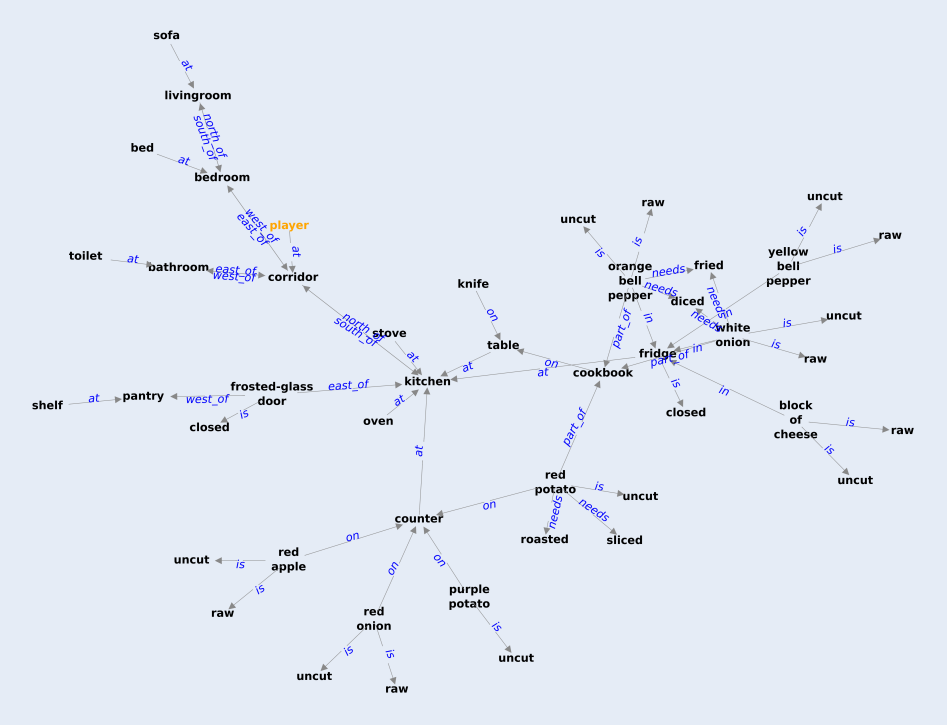


Figure 18: $\mathcal{G}^{\text{full}}$ at the start of a *FTWP* game.