Scene-aware Activity Program Generation with Language Guidance Supplementary Material

ZEJIA SU, Shenzhen University, China
QINGNAN FAN, Tencent AI Lab, China
XUELIN CHEN, Tencent AI Lab, China
OLIVER VAN KAICK, Carleton University, Canada
HUI HUANG, Shenzhen University, China
RUIZHEN HU*, Shenzhen University, China

ACM Reference Format:
Zejia Su, Qingnan Fan, Xuelin Chen, Oliver van Kaick, Hui Huang, and Ruizhen Hu. 2023. Scene-aware Activity Program Generation with Language Guidance Supplementary Material. *ACM Trans. Graph.* 42, 6 (December 2023), 5 pages. https://doi.org/10.1145/3618338

1 METHOD DETAILS

Prompt Template. We explain the construction of the category-aware long prompt in more detail here, which is illustrated in Figure 1. We first translate each instruction A^k in the program $\mathcal{P}^t$ into a template sentence A_c^k by filling the category labels of the involved objects into a pre-defined template for the action. Then, the category-aware long prompt is generated by filling the translated instruction sentences $\mathcal{P}_c^t = \{A_c^k\}_{k=1}^t$ and the description $\mathcal{D}$ into a prompt template with keywords that identify the description and the program.

Feature propagation in the aHGN. The aHGN is composed of multiple layers where the feature propagation is performed layer per layer. Given a set of node features $M^{t,(l)} = \{m_i^{t,(l)}\}_{i=1}^{N_v}$ of layer l, the adjacent objects $\mathrm{AD}^t = \{\mathrm{ad}_i^t\}_{i=1}^{N_v}$ of the nodes, and the global features F_g^t and F_l^t, each node feature $m_i^{t+1,(l+1)}$ at layer $l+1$ is updated with its incoming information $\delta_i^{t+1,(l+1)}$, which is produced by aggregating the node features of the node's neighbors $j \in \mathrm{ad}_i^t$ at the previous layer:

$$m_i^{t+1,(l+1)} = \mathbf{GRUU}(m_i^{t+1,(l)}, \delta_i^{t+1,(l+1)}), \quad \text{with} \tag{1}$$

$$\delta_i^{t+1,(l+1)} = \|_{k=1}^{K} \delta_{ik}^{t+1,(l+1)}, \tag{2}$$

*Corresponding author: Ruizhen Hu (ruizhen.hu@gmail.com).

Authors' addresses: Zejia Su, zejiasu.36@gmail.com, Shenzhen University, China; Qingnan Fan, fqnchina@gmail.com, Tencent AI Lab, China; Xuelin Chen, xuelin.chen.3d@gmail.com, Tencent AI Lab, China; Oliver van Kaick, ovankaic@gmail.com, Carleton University, Canada; Hui Huang, hhzhiyan@gmail.com, Shenzhen University, China; Ruizhen Hu, ruizhen.hu@gmail.com, Shenzhen University, China.

0730-0301/2023/12-ART $15.00
https://doi.org/10.1145/3618338

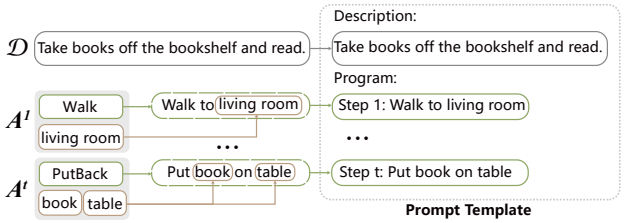

Fig. 1. Example of the process for generating a long prompt for a description and partial program.

$$\delta_{ik}^{t+1,(l+1)} = \sum_{j \in \mathrm{ad}_i^t} w_{ijk}^{(l+1)} \mathbf{MLP}(m_j^{t+1,(l)}), \tag{3}$$

$$w_{ijk}^{(l+1)} = (1-\beta)\hat{w}_{ijk}^{(l+1)} + \beta w_{ijk}^{(l)}, \quad \text{and} \tag{4}$$

$$\hat{w}_{ijk}^{(l+1)} = \mathbf{softmax}_j(\mathbf{MLP}([m_i^{t+1,(l)}, m_j^{t+1,(l)}, F_g^t, F_l^t, \mathcal{E}_e(e_{ij})])), \tag{5}$$

where **GRUU** refers to a GRU update module like the one proposed by Li et al. [2015]. K denotes the number of heads of the multi-head attention [Vaswani et al. 2017], $\|$ denotes the concatenation operation, β is a scaling factor for the residual connection of attention scores between layers [He et al. 2020], $\mathcal{E}_e(e_{ij})$ is the relation embedding for the relation type e_{ij}, and $\mathbf{softmax}_j$ indicates performing **softmax** over all the adjacent objects $j \in \mathrm{ad}_i^t$ of object i.

Loss Function. Our network is trained end-to-end with the loss function defined for each iteration of instruction generation as follows:

$$L = \omega_a L_a + \omega_f L_f + \omega_c L_c + \omega_h L_h, \tag{6}$$

where L_a is the action loss, $L_f = L_f^1 + L_f^2$ is the fused instance loss, $L_c = L_c^1 + L_c^2$ is the category loss, and L_h is the human link loss. We provide more details on these terms as follows.

Given the ground truth instruction $\hat{A}^{t+1} = (\hat{a}^{t+1}, \hat{v}_{i\_1}^{t+1}, \hat{v}_{i\_2}^{t+1})$ and the corresponding category labels $(c_{i\_1}^{t+1}, c_{i\_2}^{t+1})$ of objects involved, L_a, L_f^1, L_c^1 are computed with the cross-entropy loss:

$$L_a = -\sum_{i=1}^{N_A} \phi(i, \hat{a}^{t+1}) log(\bar{P}_{a,i}^t), \quad \bar{P}_a^t = \mathbf{softmax}(P_a^t), \tag{7}$$

$$L_f^1 = -\sum_{i=1}^{N_V} \phi(i, \hat{v}_{i\_1}^{t+1}) log(\bar{P}_{f,i}^t), \quad \bar{P}_f^t = \mathbf{softmax}(P_f^t), \tag{8}$$

$$L_c^1 = -\sum_{i=1}^{N_C} \phi(i, \hat{v}_{c\_1}^{t+1}) log(\bar{P}_{c,i}^t), \quad \bar{P}_c^t = \textbf{softmax}(P_c^t), \tag{9}$$

where $\phi(index, label)$ denotes a binary indicator which returns *True* only if *index* is equal to *label*. L_f^2 and L_c^2 are computed in the same way as L_f^1 and L_c^2, respectively.

The human-link loss is computed with binary cross-entropy loss:

$$L_h = \frac{1}{N_V} \sum_{i=1}^{N_V} \psi_h^t(i) log(\bar{P}_{h,i}^t) + (1 - \psi_h^t(i)) log(1 - \bar{P}_{h,i}^t), \quad \textbf{where } \bar{P}_{h,i}^t = \textbf{sigmoid}(P_{h,i}^t), \tag{10}$$

with $\psi_h^t(i)$ denoting a binary indicator for the future link of the human node, which returns *True* only if there exists a spatial link between node v_i^{t+1} and the human agent node at time step $t+1$.

2 IMPLEMENTATION DETAILS

Hyper-parameters and network architecture details. The data comprises 45 actions, 284 object categories, and 5 relation types. The number of dimensions of the embedding vectors, including the action embedding, category embedding, word embedding, and relation embedding, is set to 300, except for the edge embedding in the aHGN, which is set to 64. All the MLPs have 256 hidden units, except for those in the *memory-based feature update* module, which have 64 hidden units, and those in the *activity-aware feature propagation* module, which have 16 hidden units. All the activation functions in the MLPs are *Tanh*, except for those in the *aHGN* which are *LeakyReLU* with a negative slope value of 0.05. In the *two-branch feature encoding* module, we adopt GPT-2 [Wei et al. 2021] as the language model, and utilize LoRA [Hu et al. 2021] to fine-tune only the newly added weights of GPT-2, where the rank of the update matrices is set to 16, the scaling factor is set to 32, and the dropout rate is set to 0.1. The word embedding is initialized with GloVe [Pennington et al. 2014], where both GRUs have 256 hidden units. In the *feature-hybrid instruction generation* module, the GRU in the computation of the *human-centric probability* has 64 hidden units. In the *aHGN* in the *instruction execution and scene update* module, the number of heads of the multi-head attention is set to 4, the GRU has 64 hidden units, and the scaling factor for the residual connection is set to 0.05.

Training Details. Our networks are implemented using PyTorch and optimized using the Adam optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The initial learning rate is 3×10^{-4}, which is multiplied by 0.9 per 1,000 iterations. We train the networks with a batch size of 8 on an NVIDIA 3090ti GPU for 20,000 iterations for almost 45 hours.

Model Details. The numbers of parameters and inference time of our method and all the baselines are shown in Table 1. Note that we perform inference for the ZP (GPT-3), ZP-S (GPT-3) and ZP-S(GPT-4) methods with and online API. Thus, their inference time cannot be measured.

ZP (GPT-3), ZP-S (GPT-3) and ZP-S(GPT-4) have the largest number of parameters, since they use the large language model GPT-3 [Brown et al. 2020] or GPT-4 [OpenAI 2023]. The number of parameters of RAG (Two, GRU) and RAG (Two, GPT-2) are both larger than RAG (One, GRU), as these methods adopt a two-stage framework. Our model also has many parameters, mainly due to the use of the pre-trained language model GPT-2 [Wei et al. 2021].

Table 1. The numbers of parameters and inference time of our method and all the baselines. The inference time is measured on an NVIDIA 3090Ti GPU with batch size of 1.

	Parameters (M)	Inference Time (s)
VH (GRU)	4.93	0.025
VH (GPT-2)	131.00	0.308
ZP (GPT-3)	175,000.00	-
ZP-S (GPT-3)	175,000.00	-
ZP-S (GPT-4)	170,000,000.00	-
RAG (Two, GRU)	129.25	0.308
RAG (Two, GPT-2)	6.83	0.325
RAG (One, GRU)	4.24	0.305
Ours	137.33	1.120

Table 2. Performance comparison of our method in training and testing. The evaluation metrics are the same as those used in the main text.

	Rationality	LCS	Executability	Completeness
Training	0.720	0.878	0.874	0.859
Run-time	0.438	0.515	0.746	0.584

We also find that VH (GRU) has the shortest inference time, because it does not use the scene graph as input, and thus it does not require time to process the scene-related data. Since we utilize a pre-trained language model and perform dynamic scene update operations, the inference time of our method is relatively long.

The performance of our method during training and at testing is shown in Table 2. We can see that the performance of our method in the two phases is relatively stable. We notice a drop in the Rationality, LCS, and Completeness scores during run-time as those three metrics require a comparison to the GT program, which is only one possible way of executing the corresponding activity.

Details of Evaluation Metrics. According to Liao et al. [2019], *Completeness* is computed between the scene graphs G and $\hat{G}$ using the $\mathbb{F}_1$ score, where G and $\hat{G}$ are the final scene graphs after the execution of the predicted and ground truth programs, respectively:

$$\mathbb{F}_1(\hat{G}, G) = \frac{2 \cdot \mathbb{P}(\hat{G}, G) \cdot \mathbb{R}(\hat{G}, G)}{\mathbb{P}(\hat{G}, G) + \mathbb{R}(\hat{G}, G)}, \quad \text{with} \tag{11}$$

$$\mathbb{P}(\hat{G}, G) = \frac{\|\mathbb{T}(\hat{G})\| \otimes \|\mathbb{T}(G)\|}{\|\mathbb{T}(\hat{G})\|}, \quad \text{and} \tag{12}$$

$$\mathbb{R}(\hat{G}, G) = \frac{\|\mathbb{T}(\hat{G})\| \otimes \|\mathbb{T}(G)\|}{\|\mathbb{T}(\hat{G})\|}, \tag{13}$$

where $\otimes$, $\mathbb{P}$, and $\mathbb{R}$ are the binary matching function, precision, and recall, respectively. $\mathbb{T}$ is a function that converts a graph to tuples. In particular, we only compare the sub-graph containing the object instances mentioned in the predicted and ground truth programs.

Prompt

Scene-aware example part 1 : Scene graph as a nested list

```
Scene_1 = [
    ("bathroom", [
        ("bedroom_cabinet", [opened], [
            ("detergent"),
        ),
        ...
        ("computer", [switched_off, plugged_out]),
    ])
    ...
    ("bedroom", [
        ("dresser", [closed], [
            ("hanger"),
        ),
        ...
        ("cat"),
        ("couch"),
        ("light", [switch_off, plugged_out]),
    ])
]
```

Scene-aware example part 2 : Program associating with scene states

```
Task_1 = {}
Task_1["Scene"] = Scene_1
Task_1["Description"] = "turn on light, pet my cat in the couch."
Task_1["Program"] = [
    # since the "bedroom" in scene_1 has "cat", "light" and "couch", we need to walk
      to it.
    ("walk", "bedroom", "None"),
    ("walk", "light", "None"),
    ("find", "light", "None"),
    # since the state of "light" in "bedroom" in scene_1 is [switch_off, plugged_out],
      we need to first plug it in and then switch on it.
    ("plugin", "light", "None"),
    ("switchon", "light", "None"),
    ("walk", "couch", "None"),
    ("sit", "couch", "None"),
    ("turnto", "cat", "None"),
    ("touch", "cat", "None"),
]
```

Other scene-aware examples

```
Scene_2 = [...]
Task_2 = {...}
Scene_3 = [...]
Task_3 = {...}
```

New Scene and Description for inference

```
Scene_4 = [...]
Task_4 = {}
Task_4["Scene"] = Scene_4
Task_4["Description"] = "Take book off the bookshelf."
```

Fig. 2. The prompt with scene-aware examples constructed for the ZP-S method.

Executability refers to the semantic correctness of each instruction in a program. For an instruction to be executable, it needs to meet the following two conditions: 1) The combination of the object and action is valid, e.g., "Grab bathroom" is invalid, and 2) The instruction can be performed under a given scene state, e.g., "Grab book" is only valid if the human agent has closed the object "book".

3 EXPERIMENT DETAILS

Prompt construction for ZP-S. The zero-shot planner [Huang et al. 2022] generates a program without taking the current scene into consideration, which may predict instructions that are inconsistent with the scene state, e.g., turn on a TV that is already on. Since the LLM generates new samples by imitating the given samples, we construct a new prompt that contains several scene-aware examples to make the LLM learn to generate a program based on the given scene. Specifically, for a given input, we first transform the corresponding scene graph into text and add it at the front of the program. Then, we modify the example program accordingly by explicitly associating some of its instructions with the scene states.

Inspired by ProgPrompt [Singh et al. 2022], we create a prompt structured as python code and use an LLM to complete the code. The prompt contains three scene-aware examples, and each example is composed of three parts: a scene graph presented in text format, an activity description, and a program associated with the scene states.

Due to the limited number of input tokens that the LLM can accept, we cannot convert the whole scene graph into text, which contains many nodes and edges. For this reason, we keep only those parts of the scene graph that are useful for program generation. For each node, we keep only the category of the object and its state, e.g., "light" and ("switched_off", "plugged_out"). For the edges between nodes, we only keep the edges of type "inside" because they are sufficient for identifying the location of an object for program generation. For example, if we would like to generate a program for the activity "read book", we only need to know where the book is, which can be clearly indicated by the edges of type "inside". Considering the above two points, we choose to use nested python lists to represent the scene graph, which contains a list of objects in the scene. Each object in the list is represented as a python tuple ("class_name", "state_list", "child_object_list"), where "class_name" is the category of the object, "state_list" is the state list of the object, and "child_object_list" is the list of related nodes with edges of type "inside". As shown in the first block of Figure 2, the scene contains a "bedroom", which contains a "bedroom_cabinet" with state "open" and a "computer" with state "switched_off" and "plugged_out", where "detergent" is inside the "bedroom_cabinet".

Besides adding the scene into the prompt, we also need to modify the program accordingly to make some of its steps consider the scene states. Specifically, we add the corresponding scene conditions before the instructions that need to be predicted based on the scene states. As shown in the second block of Figure 2, we add the sentence "since the light is plugged in and switched off" in the form of a code *comment* before the instructions "plug in light" and "switch on light". Given an example program like this, the LLM can learn to generate instructions based on the scene state by first predicting a corresponding *comment* with scene information. To further facilitate the parsing of the sentences generated by the LLM into the formatted instructions, each instruction in the example program is represented as a tuple (action, object1, object2), and thus the LLM can generate instructions in such a format, where the nouns of the action and objects can be simply extracted from the sentences.

Based on the given examples, the LLM can learn to imitate to generate the program for new examples using the corresponding

Table 3. Statistics of error types that lead to a program not being executable.

AC	DU	LG	Attribute	Distance	State	Total
			632	568	175	1375
		✓	178	639	232	1049
	✓		311	400	**33**	744
✓			528	**336**	108	972
✓	✓		346	393	53	792
✓	✓	✓	**145**	412	139	**696**

Table 4. Ablation study on language guidance.

	Rationality	LCS	Executability	Completeness
Strategy 1	0.380	0.443	0.515	0.366
Strategy 2	0.384	0.425	0.548	0.415
Strategy 3	**0.408**	**0.459**	0.566	0.428
Strategy 4	0.391	0.451	**0.600**	**0.455**

scene information. Given a new scene graph and description as input, we first transform the new scene into the nested list described above, fill the list and the description into the prompt, and then feed the entire prompt to the LLM. The LLM will generate a complete program with many sentences, and the instructions can be simply parsed from these sentences by matching the extracted words to those in the action set and object category set based on semantic similarity, like the operations on the Zero-shot Planner [Huang et al. 2022].

Analysis of executability in the results. We further analyzed the impact of each module on the executability of the programs generated by our method. We counted different types of errors that lead to non-executability as a percentage of the total test cases, including attribute-oriented errors, state-oriented errors, and distance-oriented errors. Attribute-oriented errors refer to the type of action performed on an object that does not match its attributes, e.g., performing a grabbing action on a couch that does not have the attribute "grabbable". State-oriented error means that the execution of a command does not satisfy the abstract scene state, e.g., performing a 'turn on" action on a TV that is already turned on. A distance-oriented error is when the command does not satisfy the interaction distance, such as grabbing an object that is not near the human agent. Note that an instruction may involve multiple errors. The results are shown in Table 3.

We find that all the key components of the method help reduce these types of errors. Among them, Language Guidance (LG) mostly reduces attribute-oriented errors because it semantically constrains the selection of objects and actions with the language priors. Dynamic graph Update (DU) mostly reduces state-oriented errors because it explicitly updates the scene and therefore the model can generate instructions based on an accurate scene state. Adjacency Constraints (AC) mostly reduce distance-oriented errors because they impose constraints on the selection of interactive objects to prevent interaction with objects that are not near the human agent in the next step.

Table 5. Ablation study on dynamic graph update.

MFU	AFP	Rationality	LCS	Executability	Completeness
		0.348	0.399	0.479	0.338
✓		0.345	0.409	0.648	0.415
	✓	0.334	0.415	0.633	0.464
✓	✓	**0.360**	**0.425**	**0.691**	**0.498**

Table 6. Ablation study on adjacency constraints by considering the human agent's position.

History	Rationality	LCS	Executability	Completeness
	0.326	0.370	0.588	0.392
✓	**0.346**	**0.423**	**0.604**	**0.423**

Ablation study on language guidance. As language guidance is one of the most important components of our method, we further analyze different strategies for introducing a PLM to guide the generation of programs:

• *Strategy 1*: use solely F_l^t extracted by the PLM to predict the instance-wise probability of objects.

• *Strategy 2*: utilize F_l^t to predict the instance-wise probability instead of the category-wise one, and fuse the probability with P_g^t predicted by F_g^t.

• *Strategy 3*: utilize F_l^t to predict the category probability P_c^t without the global scene features $F_{\mathcal{S}}^t$, and fuse the probability with P_g^t predicted by F_g^t.

• *Strategy 4*: utilize F_l^t, along with the global scene features $F_{\mathcal{S}}^t$, to predict the category probability P_c^t, and fuse the probability with P_g^t predicted by F_g^t.

As shown in Table 4, the performance of *Strategies 3* and *4* that utilize the PLM to predict category probabilities is obviously better than *Strategies 1* and *2* that utilize the PLM to predict instance-wise probabilities. These results demonstrate that it is better to utilize the feature F_l^t with semantic information to predict the semantic category of objects than the other options. Moreover, *Strategy 4* performs better than *Strategy 3* in terms of *Executability* and *Completeness*, proving the importance of global scene features $F_{\mathcal{S}}^t$ induced from the graph-guided probability P_t^g.

Ablation study on dynamic graph update. We also analyze the sub-modules that perform the dynamic graph update, including the memory-based feature update (MFU) and activity-aware feature propagation (AFP). The results are shown in Table 5. We see that both sub-modules can solely improve the performance.

Ablation study on adjacency constraints. We further analyze the effectiveness of the human agent's historical information for adjacency constraints. The baseline without the historical information is created by replacing the historical human feature F_h^t with the latest node feature of the human agent z_h^t to predict the human-centric probability. The results are shown in Table 6. We can see that the introduction of the historical information of the human agent helps to improve the performance.

REFERENCES

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

Ruining He, Anirudh Ravula, Bhargav Kanagal, and Joshua Ainslie. 2020. Realformer: Transformer likes residual attention. *arXiv preprint arXiv:2012.11747* (2020).

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *arXiv preprint arXiv:2201.07207* (2022).

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

Yuan-Hong Liao, Xavier Puig, Marko Boben, Antonio Torralba, and Sanja Fidler. 2019. Synthesizing environment-aware activities via activity sketches. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 6291–6299.

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2022. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302* (2022).

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).