**proScript: Partially Ordered Scripts Generation**

Keisuke Sakaguchi, 1 Chandra Bhagavatula, 1 Ronan Le Bras, 1 Niket Tandon, 1 Peter Clark, 1 Yejin Choi 1, 2
1 Allen Institute for Artificial Intelligence
2 Paul G. Allen School of Computer Science & Engineering, University of Washington

**Abstract**

Scripts – prototypical event sequences describing everyday activities – have been shown to help understand narratives by providing expectations, resolving ambiguity, and filling in unstated information. However, to date they have proved hard to author or extract from text. In this work, we demonstrate for the first time that pre-trained neural language models can be finetuned to generate high-quality scripts, at varying levels of granularity, for a wide range of everyday scenarios (e.g., bake a cake). To do this, we collect a large (6.4k) crowdsourced partially ordered scripts (named proScript), that is substantially larger than prior datasets, and develop models that generate scripts by combining language generation and graph structure prediction. We define two complementary tasks: (i) edge prediction: given a scenario and unordered events, organize the events into a valid (possibly partial-order) script, and (ii) script generation: given only a scenario, generate events and organize them into a (possibly partial-order) script. Our experiments show that our models perform well (e.g., F1=75.7 on task (i)), illustrating a new approach to overcoming previous barriers to script collection. We also show that there is still significant room for improvement toward human level performance. Together, our tasks, dataset, and models offer a new research direction for learning script knowledge.

1 Introduction

Scripts (Schank and Abelson, 1975) represent structured commonsense knowledge about prototypical events in everyday situations/scenarios such as bake a cake (Figure 1). However, while scripts have been shown to help understand narratives by providing expectations, resolving ambiguity, and filling in unstated information (Chambers and Jurafsky, 2008; Modi et al., 2017, inter alia), they have proved hard to author or extract from text, with only small script databases available (Regneri et al., 2010; Chambers, 2017; Ostermann, 2020).

In this work, we show for the first time that pre-trained neural language models (LMs) can be adapted to generate high-quality scripts, including appropriately partial ordering events where a specific temporal ordering is required only when it is necessary. LMs have previously been shown to successfully generate stories (Rashkin et al., 2020), summaries (Lewis et al., 2020), and commonsense facts (Bosselut et al., 2019; Hwang et al., 2020). Here we investigate their application to script generation. First, we collect large amount (6.4k) of partially ordered scripts from crowdsourcing with a similar but simplified collection method (Ciosici et al., 2021). We call the dataset as proScript (PaRtial Order SCRIPT), and this is substantially larger and more diverse than prior (crowdsourced) dataset such as DeScript (Regneri et al., 2010) that has 40 scripts. In proScript, all the events/paths need to be happened/executed (cf. AND arcs in AND/OR graphs), whereas prior work on scripts...
do not distinct core and optional/alternative events explicitly. Additionally, temporal duration of each event is annotated (e.g., *take the cake out of the oven* typically takes one minute in the *bake a cake* script), which will potentially link script knowledge with temporal reasoning in future work.\(^1\)

Second, with the collected data, we introduce two complementary tasks: **script edge prediction** and **entire script generation**. In the edge prediction task, given a scenario and unordered intermediate events, models must organize the events as a valid partial-order script. On the other hand, the script generation task is to generate intermediate events and the partial-order of those events for a given scenario. This task requires both natural language generation (for nodes) and graph structure prediction (for edges).

Finally, based on our proposed dataset, we develop models for both edge prediction and entire script generation tasks. As Chambers (2017) has revealed that models trained and evaluated on missing events prediction (i.e., *narrative cloze*) are insufficient to assess script knowledge, our evaluation scheme evaluate the entire script. We compare the models against baselines, and show that our models outperform the baselines for both the edge prediction and the script generation tasks. Nonetheless, there is a significant room for improvement toward human-level performance – e.g., for edge prediction, the best model achieves 75.71 of F1 score while human achieves 89.28, and for script generation, the best model obtains a graph edit distance of 4.97 (i.e., number of human edits), while human-created scripts achieve 2.98 on average.

Our contributions are thus:

- A new dataset (**proScript**) of crowdsourced scripts that is substantially larger than prior (manually crafted) datasets
- Two complementary task definitions against **proScript**
- Two new models for these task, providing the first demonstration that generative models can be successfully applied, although it is still significantly below human levels.

## 2 Related Work

### Script as narrative chain

Mooney and DeJong (1985) and Chambers and Jurafsky (2008, inter alia) have investigated automatically inducing scripts from (unstructured) corpus. In particular, Chambers and Jurafsky (2008) introduced scripts as *narrative chain*, where verbs with the participants information (e.g., (*claimed, subj*), and (*accused, obj* )) named *narrative events* are partially ordered according to causal and temporal relations. They also introduced *narrative cloze* task, where a model is expected to predict one removed narrative event, given all the other narrative events, while our proposed task requires to generate scripts as a partial-order graph for a given scenario. The “script as narrative chain” approach has been actively studied (Jans et al., 2012; Modi and Titov, 2014; Pichotta and Mooney, 2014; Rudinger et al., 2015; Granroth-Wilding and Clark, 2016; Weber et al., 2018; Belyy and Van Durme, 2020), but it has its drawbacks. First, the source corpora is mainly from a news domain rather than everyday scenarios, and induced narrative chains contain a number of non-script events such as reporting verbs (Mostafazadeh et al., 2016; Chambers, 2017). Second, events are highly abstracted as tuples of verb and the dependency (*subj* or *obj*) (Ostermann, 2020). Third, the evaluation scheme for the narrative cloze task is insufficient to evaluate script knowledge (Chambers, 2017).

### Script as paraphrase sets

Script as paraphrase sets (Regneri et al., 2010; Modi et al., 2016; Wanzare et al., 2016) is more recent approach to gather script knowledge, where crowd workers are asked to write down a sequence of events for a given everyday scenario (e.g., bake a cake) and the collected sequences (called event sequence description) are aligned with paraphrased events being clustered. The collected (partially ordered) scripts cover wide variety of everyday situations compared to narrative chains (news domain), but one shortcoming of this approach is the scalability; it is not easy to scale because of the cost for manual data collection (Chambers, 2017; Ostermann, 2020). In fact, Modi et al. (2016) crowdsourced 1000 stories that cover only 10 scripts, and similarly Regneri et al. (2010) end up with collecting 40 scripts. The limited amount of data hinders learning script knowledge by models. Furthermore, they provide no evaluation metric on the dataset for assessing model’s script knowledge.

### Story generation and tracking state changes

Neural models have been demonstrated to successfully generate stories (Kiddon et al., 2016; Peng et al., 2018; Zhai et al., 2019; Rashkin et al., 2020)
as well as tracking state changes in procedural texts (Henaff et al., 2017; Bosselut et al., 2018; Dalvi et al., 2018; Tandon et al., 2020). Our work is related in terms of generating higher-level agenda (or plot) of a story and understanding latent preconditions and effects between events. However, a main difference between these studies and scripts is that story generation and state change tracking explicitly generate and/or predict character’s mental states and entity’s physical attributes (e.g., temperature), whereas scripts focuses on essential core events (Chambers, 2017) in partial order.

3 Definitions

**proScript** We define **proScript** as a directed acyclic graph (DAG), $G(V, E)$ with a given scenario $(s)$, where $V$ is a set of essential events \( \{ v_1, \ldots, v_i, \ldots, v_j \} \) and $E$ is a set of temporal ordering constraints between events $\{ e_{ij} \}$ which means that the events $v_i$ must precede the event $v_j$ ($v_i \prec v_j$). DAGs effectively encode the partial-ordering of core events—crucial for representing events which can be performed in any order. For example, in a *bake a cake* scenario, one can “gather the ingredients” and “turn on the oven” in any order (Figure 1). We emphasize that scripts should not include non-core events such as discourse related events (e.g., reporting verbs) as Chambers (2017) proposed. In **proScript**, we also exclude alternative events in a **proScript** DAG. For example, in a *bake a cake* scenario, “get ingredients” and “buy ingredients” are alternative events with each other because either one is only necessary in the scenario. By excluding alternative events, we can resolve ambiguity of the edges in partial order structure as temporal relations or alternative paths. Regneri et al. (2010) and Modi et al. (2016) do not discriminate this ambiguity.

With the definition, we introduce **proScript** task in two complementary settings: script edge prediction and entire script generation.

**Edge Prediction** The script edge prediction task is to predict a set of partial-ordered edges $(E)$ of the script $G(V, E)$, given a scenario and a set of unordered intermediate events $v \in V$.

2Technically, **proScript** is a transitive reduction of a DAG. In short, transitive reduction of $G$ does not have any short cut edges between nodes. In **proScript**, we add a single root node $(v_r)$ and scenario $(s)$ as a unique leaf node.

3We focus on events and the partial-ordering for the *protagonist* (Chambers and Jurafsky, 2008), and leave the identification of other participants for future work.

---

Suppose a scenario where someone wants to “bake a cake”.

<table>
<thead>
<tr>
<th>Preliminary Question:</th>
</tr>
</thead>
<tbody>
<tr>
<td>How long will it take for this scenario?</td>
</tr>
</tbody>
</table>

- [ ] second(s)
- [ ] minute(s)
- [ ] hour(s)
- [ ] day(s)
- [ ] month(s)
- [ ] year(s)

<table>
<thead>
<tr>
<th>Main Question 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Describe 5 to 7 essential steps and each time duration.</td>
</tr>
</tbody>
</table>

- **find the cake recipe**: 10 minute(s)
- **gather the ingredients**: 15 minute(s)
- **turn on the oven**: 2 minute(s)

<table>
<thead>
<tr>
<th>Main Question 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a flowchart of the steps (possibly in partial order, where temporal ordering is required only when it is necessary).</td>
</tr>
</tbody>
</table>

![Figure 2: Annotation procedure for proScript data collection.](image)

**Script Generation** The script generation task is to predict a partial order script $G(V, E)$, but only the scenario is given. Models are additionally expected to generate events $(V)$ in natural language.

4 Datasets

**Source of Scenarios** We collected scenarios from DeScript (Wanzare et al., 2016), VirtualHome (Puig et al., 2018), and ROCStories (Mostafazadeh et al., 2016). DeScript consists of 40 daily scenarios (e.g., making coffee) and we
use all of them. VirtualHome is constructed to learn activities interactively in a household in a 3D simulated world. It has 233 indoor tasks (e.g., turn on light) and we include them as scenarios. Since these two datasets have only small amount of scenarios, we additionally extracted phrases for scenarios from ROCStories (Mostafazadeh et al., 2016), by manually curating patterns with want(ed) to ... (e.g., go to Hawaii), need(ed) to ... (e.g, get a haircut) and look(ing) to (e.g, buy a television). The scenarios we collected from ROCStories include both high-level long-term ones (e.g., open a small business) and fine-grained short-term ones (e.g., sign into an email account).

Crowdsourcing proScript For the collected scenarios, we crowdsource the corresponding proScript on the Amazon Mechanical Turk. Our crowdsourcing procedure (Figure 2) is similar but simplified method to (Ciosici et al., 2021). First, given a scenario (e.g., bake a cake), each crowdworker is required to describe five to seven core events that they are essential for the scenario (Chambers, 2017) with the estimated time it takes to complete each event. In the second question, crowdworkers confirm the set of steps and they are asked to create a flowchart (DAG) by connecting the steps possibly in partial order. When crowdworkers make a submission, validation function is executed to check if the created flowchart is a valid (transitive reduction of) DAG that does not contain a cycle/loop and any short cut edge.

Due to the complex nature of this crowdsourcing procedure, it is crucial to maintain the quality. To filter out noisy scripts and resolve conflicts, two different workers are asked to sort the same set of events in partial order (i.e., the same as the second question described above), and we filter out scripts (DAGs) that have low agreement. To collect proScript with both micro and macroscopic scenarios, we iteratively picked events in the DAGs and use them as an additional source of finer-grained scenarios. For example, turn on the oven is a new fine-grained scenario derived from bake a cake.

### Dataset Statistics
In total, we collected 6,414 valid scripts that include 311,502 pairs of events, and we split the proScript into training (3,252 scenarios), development (1,085), and test set (2,077). The training and development sets consist of scenarios collected from ROCStories, and the test set consists of those from ROCStories, Describe, and VirtualHome. This helps us evaluate in- and out-of-domain performance.

<table>
<thead>
<tr>
<th>source</th>
<th>train</th>
<th>dev</th>
<th>test (in)</th>
<th>test (out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>scenarios</td>
<td>ROC</td>
<td>ROC</td>
<td>ROC VirtualHome</td>
<td>971</td>
</tr>
<tr>
<td>scenes</td>
<td>3,252</td>
<td>1,085</td>
<td>1,106</td>
<td>971</td>
</tr>
</tbody>
</table>

The average number of events in proScript scenarios is 5.45 and the maximum degrees of DAGs in the training set are distributed as follows: 2,198 scripts (67.6%) for degree 1, 915 scripts (28.1%) for degree 2, 108 scripts (3.3%) for degree 3, 31 scripts (0.9%) for degree 4 and above.

Figure 3 shows the normalized histogram of the typical time to take for each script in proScript dataset. Most of the scripts take between a minute and an hour (e.g., “go to bathroom”, “buy some new clothes”), while there are a reasonable amount of high-level long-term scripts (e.g., “find a new job”, “open a small business”).

### 5 proScript Edge Prediction

#### 5.1 Models
For the proScript edge prediction task (§3), we implement a two-step approach baseline (pairwise model) and compare it with our proposed end-to-end neural method (proScript_edge-pred).

**Pairwise Model** We implement a two-step baseline where we train a binary classifier to predict the
step 0: turn on the oven;
step 1: bake for the right amount of time;
step 2: mix the ingredients;
step 3: find the cake recipe;
step 4: gather the ingredients;
step 5: put the cake batter in the oven;
step 6: take the cake out of the oven;

Figure 4: Example of input and output for the proScript edge-pred model. The input is a flattened sequence of events, and the output is a flattened sequence of edges of the (predicted) partial-order script in DOT language (Gansner et al., 1993).

precedence between pairs of events, followed by building a partial order script \( G \) by aggregating the predicted relations across all pairs of events.

Formally, the classifier takes a pair of events \((v_i, v_j)\) and predicts the precedence \( e_{ij} \) i.e. the event \( v_i \) precedes \( v_j \).

\[ e_{ij} = p(v_i \prec v_j | v_i, v_j) \quad (1) \]

Scores by the classifier are used as weights to create an adjacency matrix of \( G \) which is then automatically converted into a partial-order script with heuristics – when \( G \) contains a cycle, we iteratively remove edges by choosing the one with minimum weight until we get a valid DAG.

proScript edge-pred We propose an end-to-end neural model, which takes all the (unordered) events \((v)\) and the scenario \((s)\) as the input \((x)\) and predicts the edges \((E)\) in a partial-order script \((G)\) at one time. To represent \( E \) in a linear format \((y)\), we use DOT, a graph description language (Gansner et al., 1993) as shown in Figure 4.7 By flattening the nodes and edges of \( G \), we apply neural encoder-decoder models. Formally, flattened unordered events and scenario as \( x \) are embedded as continuous representation \((\text{emb}(x))\) by the encoder, then the decoder will generate tokens \((y)\) as follows:

\[
p(y_1, \ldots, y_N | x_1, \ldots, x_M) = \prod_{n=1}^{N} p(y_n | \text{emb}(x_1, \ldots, x_M), y_1, \ldots, y_{n-1}).
\]

Compared to the pairwise model, the proScript edge-pred model uses information from all the events jointly to build partial-order script with a broader context.

5.2 Evaluation Metrics
Given \( \hat{G}(V, \hat{E}) \) as a predicted (partial order) script and \( G(V, E) \) as the correct (oracle) script, the F1 score is defined as follows:

\[
\text{Precision} = \frac{|E \& \hat{E}|}{|E|}, \quad \text{Recall} = \frac{|E \& \hat{E}|}{|\hat{E}|}
\]

\[
\text{F1 score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}.
\]

For evaluating human performance, we show randomly shuffled steps to crowdworkers and ask them to create a partial-order script. We compute the F1-score of the script with the reference script.

5.3 Experiments
Setup For the binary classifier (pairwise model), we use two variants of the Transformer (Vaswani et al., 2017): RoBERTa-large (Liu et al., 2019) and T5-11B (Raffel et al., 2020).

When training (i.e., fine-tuning) RoBERTa,8 we use a grid-search for choosing the best hyper-parameters from the best performed model on the development set: epochs \{1, 2, 3\}, learning rate \{1e-5, 1e-6, 1e-7\}, batch size \{16, 24, 32\}. For training the T5 model as the pairwise model, we followed a default set of hyper-parameters that are recommended in Raffel et al. (2020).9

For the proScript edge-pred model, we use the T5 with different model sizes (Large and 11B) and training sizes (100, 1k, and all 3.2k) to see how these factors affect the performance.10 We followed a default set of hyper-parameters for the T5 models.

Results The results are shown in Table 1. We find that the pairwise and proScript edge-pred models significantly outperform the random baseline where the edges are randomly assigned. The proScript edge-pred T5-11B model outperforms the pairwise T5-11B model. This indicates that the proScript edge-pred model benefits from a larger context from the input to predict edges more accurately, although there is still a significant room for improvement toward human-level performance.11 Regarding the difference between

---

8We used the implementation from Huggingface Transformers (Wolf et al., 2019).
9https://github.com/google-research/text-to-text-transfer-transformer
10We also used BART (Lewis et al., 2020) and GPT2 (Radford et al., 2019) as the baselines, but we found that both failed to generate canonical DOT language.
11We find that 99% of the outputs from proScript edge-pred are valid DOT language.
in and out of domain, we find that the in-domain performance is higher than the out-of-domain performance, whereas human performance is robust regardless of the domain difference. We also see that the training set (100, 1k, all) and model sizes (Large, 11B) significantly affect the performance of proScript\textsubscript{edge-pred}.

Figure 5 shows the performance of the pairwise (T5-11B) model, proScript\textsubscript{edge-pred} (T5-11B) and human according to the (maximum) degree of the script DAGs. We find that scripts with higher degree are more difficult to predict for both proScript\textsubscript{edge-pred} and pairwise models, whereas human shows smaller decrease for predicting higher-degree scripts.

### 6 proScript Generation

#### 6.1 Models

**proScript\textsubscript{gen}** The proScript generation task combines natural language generation (i.e. generating events in natural language) with graph structure prediction over the generated events (i.e. organizing the events into a DAG). Our approach (proScript\textsubscript{gen}) is to formulate it as an end-to-end problem, similar to the proScript\textsubscript{edge-pred} for the proScript edge prediction task (§5.1).

![Figure 6: Example of input and output for proScript\textsubscript{gen}](https://www.wikihow.com/)

Given a scenario ($s$) and the number of events to generate in the script, proScript\textsubscript{gen} generates events and edges for the partial-order script ($G$) in DOT language (Figure 6). Formally, we use the same encoder-decoder framework (eq.2) except that a scenario and number of steps to generate are described in natural text as $x$ and the decoder is expected to generate both events and the edges (as $y$) in the script jointly.

**Transfer learning from WikiHow data** Transfer learning often helps improve the performance when it is (pre-)trained on a similar task (Peters et al., 2018; Devlin et al., 2019). As additional resource for pre-training proScript\textsubscript{gen}, we use procedural texts extracted from WikiHow,\(^\text{12}\) which contains 130k instances of a sequence of essential steps for a given topic in various categories (e.g., health, finance, hobbies, etc.). It is important to note that all the procedures in WikiHow are formatted as *sequences* rather than a partial-order, and therefore the model is biased towards generating sequences. We refer to this approach as proScript\textsubscript{gen}-transfer.

**Pipeline approach** An alternative approach is to use proScript\textsubscript{gen} followed by the

\(^{12}\)https://www.wikihow.com/
proScript-edge-pred model. The approach relies on proScript-gen to generate a set of events but allows to fix the predicted edges via the proScript-edge-pred model. We refer to this approach as proScript-gen-pipe, and study whether it can improve the performance over proScript-gen.

### 6.2 Evaluation Metrics

Chambers (2017) emphasizes the importance of human annotation for evaluating script knowledge. However, human evaluation for the proScript generation task is challenging because it involves natural text generation and graph structure prediction. As in the text generation tasks such as machine translation and text summarization, there are several possible correct answers. Therefore, we use two complementary evaluation metrics for the proScript generation task: (i) graph edit distance, and (ii) pairwise comparison. These are the absolute and relative measures of performance, respectively. Graph edit distance (Abu-Aisheh et al., 2015) computes the distance between two graphs. Formally, given two graphs $G_1$ and $G_2$,

$$\text{GED}(G_1, G_2) = \min_{\mathcal{d}_1, \ldots, \mathcal{d}_k} \sum_{i=1}^{k} \text{cost}(d_i)$$

where $d_1, \ldots, d_k$ is a list of graph edit operations from $G_1$ to $G_2$. The operations include deletion, insertion, and replacement for vertex and edge. Each operation has its cost and we set the cost to be 1 for all the operations in our evaluation for simplicity. We use an averaged graph edit distance between a model-generated script and the revised scripts by two human annotators. For evaluating human performance, a crowdworker writes a partial-order script, given a scenario. Then, similarly to the model evaluation, two human annotators are asked to revise the partial-order script, and we take the average of the two graph edit distances.

The graph edit distance is indicative of the quality of the generated scripts; higher-quality scripts must have smaller graph edit distances to the gold-standard (i.e. they require a smaller number of human revisions).

For the relative measure, we employ pairwise human judgments where we ask annotators to compare a pair of scripts generated by proScript-gen with those from the other approaches.

### 6.3 Experiments

**Setup** For our proScript-gen, we use T5-11B. Similarly to the proScript-edge-pred, we follow the default set of hyper-parameters recommended in (Raffel et al., 2020). For proScript-gen-transfer, we pre-train the proScript-gen with the 130k procedures, and finetune it on the proScript dataset. For the proScript-gen-pipe, we first obtain the actions generated by proScript-gen (ignoring the edges), and use the set of events as input for proScript-edge-pred, which is trained (see §5.3) to predict the edges.

As defined in §3, we use graph edit distance and pairwise judgments to evaluate the quality of the generated scripts. For computing graph edit distances, we select 500 scripts (250 for dev and test sets) and ask crowdworkers to revise the generated scripts as necessary (e.g., add/delete/replace the events and the edges). We use the revised scripts as gold-standard. Each script is revised by two annotators, and we compute the average of the graph edit distances.

In pairwise judgments, we compare the scripts generated by proScript-gen with those from the other approaches. We randomly select 150 pairs, and ask three crowdworkers to judge whether the script generated by proScript-gen is better, worse, or equal to the other (i.e. transfer, pipeline, or human). We use majority vote to decide the final pairwise human judgment between the two scripts.

**Results** The pairwise judgment result is shown in Figure 7. We see that the pipeline and transfer models show slight preference over the proScript-gen (except pipeline-dev), although the difference is not large. We also see that the transfer model constantly have more preference over the proScript-gen than the pipeline model in both dev and test sets. Regarding the pairwise comparison with human-created plans, proScript-gen still has a significant room for im-

![Figure 7: Pairwise judgments (%) between proScript-gen and the other approaches.](image-url)
Table 2: Results for proScript generation task (dev, test, in-domain test and out-of-domain test set). We measure the average graph edit distance between generated script and the two human revisions (lower the better). We also show the average number of each graph edit operation (\{Delete, Insert, Replace\} x \{Vertex, Edge\}). Random (edge) baseline shows 11.06 edit distance for the dev set and 10.95 for the test set.

<table>
<thead>
<tr>
<th>Split</th>
<th>Models</th>
<th>Graph Edit Dist</th>
<th>V-Del</th>
<th>V-Ins</th>
<th>V-Rep</th>
<th>E-Del</th>
<th>E-Ins</th>
<th>E-Rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>proScript_gen</td>
<td>4.73</td>
<td>0.426</td>
<td>0.192</td>
<td>0.581</td>
<td>1.558</td>
<td>1.308</td>
<td>0.671</td>
</tr>
<tr>
<td></td>
<td>proScript_gen-transfer</td>
<td>4.79</td>
<td>0.337</td>
<td>0.195</td>
<td>0.679</td>
<td>1.491</td>
<td>1.281</td>
<td>0.775</td>
</tr>
<tr>
<td></td>
<td>proScript_gen-pipe</td>
<td>4.88</td>
<td>0.397</td>
<td>0.159</td>
<td>0.560</td>
<td>1.705</td>
<td>1.407</td>
<td>0.661</td>
</tr>
<tr>
<td></td>
<td>Human</td>
<td>2.78</td>
<td>0.155</td>
<td>0.161</td>
<td>0.144</td>
<td>1.235</td>
<td>1.011</td>
<td>0.199</td>
</tr>
<tr>
<td>test</td>
<td>proScript_gen</td>
<td>4.97</td>
<td>0.581</td>
<td>0.142</td>
<td>0.656</td>
<td>1.668</td>
<td>1.184</td>
<td>0.709</td>
</tr>
<tr>
<td></td>
<td>proScript_gen-transfer</td>
<td>5.38</td>
<td>0.438</td>
<td>0.213</td>
<td>0.775</td>
<td>1.713</td>
<td>1.402</td>
<td>0.835</td>
</tr>
<tr>
<td></td>
<td>proScript_gen-pipe</td>
<td>5.41</td>
<td>0.594</td>
<td>0.143</td>
<td>0.671</td>
<td>1.880</td>
<td>1.292</td>
<td>0.787</td>
</tr>
<tr>
<td></td>
<td>Human</td>
<td>2.98</td>
<td>0.168</td>
<td>0.149</td>
<td>0.150</td>
<td>1.276</td>
<td>1.072</td>
<td>0.189</td>
</tr>
<tr>
<td>test (in domain)</td>
<td>proScript_gen</td>
<td>4.57</td>
<td>0.513</td>
<td>0.158</td>
<td>0.633</td>
<td>1.471</td>
<td>1.108</td>
<td>0.687</td>
</tr>
<tr>
<td></td>
<td>proScript_gen-transfer</td>
<td>5.03</td>
<td>0.339</td>
<td>0.299</td>
<td>0.649</td>
<td>1.575</td>
<td>1.496</td>
<td>0.677</td>
</tr>
<tr>
<td></td>
<td>proScript_gen-pipe</td>
<td>5.10</td>
<td>0.561</td>
<td>0.147</td>
<td>0.630</td>
<td>1.765</td>
<td>1.217</td>
<td>0.744</td>
</tr>
<tr>
<td></td>
<td>Human</td>
<td>3.03</td>
<td>0.168</td>
<td>0.211</td>
<td>0.154</td>
<td>1.223</td>
<td>1.091</td>
<td>0.206</td>
</tr>
<tr>
<td>test (out domain)</td>
<td>proScript_gen</td>
<td>5.43</td>
<td>0.659</td>
<td>0.124</td>
<td>0.681</td>
<td>1.894</td>
<td>1.270</td>
<td>0.735</td>
</tr>
<tr>
<td></td>
<td>proScript_gen-transfer</td>
<td>5.76</td>
<td>0.549</td>
<td>0.115</td>
<td>0.916</td>
<td>1.867</td>
<td>1.296</td>
<td>1.013</td>
</tr>
<tr>
<td></td>
<td>proScript_gen-pipe</td>
<td>5.81</td>
<td>0.659</td>
<td>0.116</td>
<td>0.795</td>
<td>1.961</td>
<td>1.267</td>
<td>0.941</td>
</tr>
<tr>
<td></td>
<td>Human</td>
<td>2.91</td>
<td>0.170</td>
<td>0.074</td>
<td>0.102</td>
<td>1.340</td>
<td>1.054</td>
<td>0.170</td>
</tr>
</tbody>
</table>

Table 2: Results for proScript generation task (dev, test, in-domain test and out-of-domain test set). We measure the average graph edit distance between generated script and the two human revisions (lower the better). We also show the average number of each graph edit operation (\{Delete, Insert, Replace\} x \{Vertex, Edge\}). Random (edge) baseline shows 11.06 edit distance for the dev set and 10.95 for the test set.

Figure 8: Histograms of graph edit distance (in dev set). The number of scripts (y-axis) according to the (binned) graph edit distance (x-axis).

Table 2 shows the average graph edit distance between the generated script and the human revisions. We find that neither transfer nor pipeline help to improve the graph edit distance over proScript_gen, indicating that proScript_gen is already a strong baseline (see examples in Appendix). The reason of no improvement by the transfer approach may be because WikiHow consists of sequences rather than partially ordered steps. No improvement by the pipeline approach indicates that the proScript_gen can directly generate valid script in both events and edges. Further studies for improvements are needed for future work.

Graph edit distance (Table 2) is absolute evaluation, and it objectively treats each graph edit equally. Pairwise comparison (Figure 7) is relative evaluation comparing generated scripts with the script quality/goodness being considered by human annotators. These two metrics produce slightly different results but this is not strictly a contradiction, as they are measuring different things.

In terms of the edit types, many of the edits are edge-related, suggesting that proScript_gen and the variants are all good at generating events but struggles with ordering them. Regarding in-and out-of domains in the test sets, we observe that proScript_gen and the variants have slightly better performance for in-domain scripts than out-of domain, while human created scripts are not affected by domains. These findings are consistent with the result in the edge prediction task (§5.3).

Figure 8 shows a histogram of the graph edit distance. It is evident that human created scripts are corrected less often than scripts generated by proScript_gen, whereas the scripts from proScript_gen and the variants often have a large number of edits (e.g., 4 or more). It is interesting to see that fewer number of scripts have 1 to 3 edits (except scripts created by human). The reason is because one simple revision tends to yield multiple graph edits (e.g., one node insertion yields multiple edge insertions).

Error Analysis  We performed manual error analysis for the scripts generated by each model. We selected 40 random scripts that have non-zero graph edit distance and classified the human revisions into
Revision types | generated script (subgraph) | revised script (subgraph)
--- | --- | ---
missing event | wait for the plane → exit the plane | wait for the plane → get on the plane → exit the plane
incorrect order | get off the car → drive to the zoo put clothes in dryer → place clothes into dryer → dry clothes | drive to the zoo → get off the car put clothes in dryer → dry clothes
irrelevant or redundant event | get a visa → get off the plane → trip to a foreign country | get off the plane → get a visa (on arrival) → trip to a foreign country
order ambiguity by context | get out of the bed → go to the kitchen | get out of the bed → open the bedroom door → go to the kitchen
granularity of events | move into new apartment | move to a new apartment
paraphrased | | 

Table 3: Examples for each revision type.

Table 4: Revision type distribution (%) by each model.

<table>
<thead>
<tr>
<th>Revision types</th>
<th>proScript</th>
<th>Transfer</th>
<th>Pipeline</th>
<th>Human</th>
</tr>
</thead>
<tbody>
<tr>
<td>crucial errors</td>
<td>(edge) incorrect order</td>
<td>15.79</td>
<td>21.62</td>
<td>24.32</td>
</tr>
<tr>
<td></td>
<td>(node) missing event</td>
<td>5.26</td>
<td>2.70</td>
<td>2.70</td>
</tr>
<tr>
<td></td>
<td>(node) irrelevant/redundant event</td>
<td>10.53</td>
<td>13.51</td>
<td>2.70</td>
</tr>
<tr>
<td></td>
<td>(edge) order ambiguity by context</td>
<td>31.58</td>
<td>32.43</td>
<td>40.54</td>
</tr>
<tr>
<td>minor revisions</td>
<td>(node) granularity of events</td>
<td>31.58</td>
<td>24.32</td>
<td>21.62</td>
</tr>
<tr>
<td></td>
<td>(node) paraphrased event</td>
<td>0.00</td>
<td>6.40</td>
<td>5.41</td>
</tr>
<tr>
<td>wrong revisions</td>
<td></td>
<td>5.26</td>
<td>5.41</td>
<td>2.70</td>
</tr>
</tbody>
</table>

7 types: (1) incorrect order of events, (2) missing event, (3) irrelevant/redundant event, (4) order ambiguity by context, (5) granularity of (core) events, (6) paraphrased event, and (7) wrong human revision/correction (examples are shown in Table 3).

Approximately, the first three error types indicate that the script has crucial errors, the next three types are trivial/minor revisions where both generated and revised scripts are plausible. The last type of revision is the one where the revised script is wrong (or worse).

Table 4 shows the statistics of each error type. We see that edge-related revisions are more frequent than node-related revisions. The generated nodes are of a high quality (among all revisions by human, 10.53% of them are related to irrelevant or redundant nodes), and the majority of revisions are minor modifications. This is consistent with the results in graph edit distance. Overall, we find that minor revisions are more frequent than crucial errors, indicating that proScript and the variants generates reasonably good scripts. In contrast, crucial errors are quite rare in human created scripts, indicating a significant room for future innovation.

7 Conclusions

We show for the first time that pre-trained neural language models can be adapted to generate partial order scripts. We collect 6,400 partially ordered script from crowdsourcing (proScript), which is substantially larger than prior manually crafted datasets. With the proScript dataset, we introduced two complementary task and models, which combine language generation and graph structure prediction, providing the first demonstration that generative models can be successfully applied to script generation, although it is still below human performance. We believe that proScript dataset and models would advance future work on various NLP tasks such as story generation, machine comprehension, temporal reasoning, and high-level planning.

References


A Appendix

A.1 Reproducibility

For training RoBERTa-large as a pairwise model, we use Quadro RTX 8000 (48GB memory), which takes around 4.5 hours to train a model. RoBERTa-large consists of 355M parameters with 24 layers, 1,024 of hidden embedding size, and 16 of the attention heads. T5-large model has 770M parameters with 24-layers, 1024-hidden-state, 4096 feed-forward hidden-state, and 16 attention heads. T5-11B models has 11B parameters with 24-layers, 1024-hidden-state, 65,536 feed-forward hidden-state, 128 attention heads. We use TPU (v3-8) on google cloud platform. It takes 3 hours in average to train a edge prediction model, and 5 hours for plan generation models.

A.2 Plans generated by proScript gen

We show some example scripts generated by proScript gen in Figure 9. In each example, proScript gen which takes scenario and the number of steps as the input (e.g., play the organ, in 5 steps) and generates a script DAG.

Figure 9: Example scripts generated proScript gen.