Knowledge Graph Generation From Text

Igor Melnyk, Pierre Dognin, Payel Das
IBM Research
Yorktown Heights, NY 10598, USA
igor.melnyk@ibm.com, pdognin@us.ibm.com, daspa@us.ibm.com

Abstract

In this work we propose a novel end-to-end multi-stage Knowledge Graph (KG) generation system from textual inputs, separating the overall process into two stages. The graph nodes are generated first using pretrained language model, followed by a simple edge construction head, enabling efficient KG extraction from the text. For each stage we consider several architectural choices that can be used depending on the available training resources. We evaluated the model on a recent WebNLG 2020 Challenge dataset, matching the state-of-the-art performance on text-to-RDF generation task, as well as on New York Times (NYT) and a large-scale TEKGEN datasets, showing strong overall performance, outperforming the existing baselines. We believe that the proposed system can serve as a viable KG construction alternative to the existing linearization or sampling-based graph generation approaches.

1 Introduction

Automatic Knowledge Graph (KG) construction is an active research area aiming at representing the information present in abundant textual corpora in a more organized, structured and compressed form, which can be efficiently utilized in a variety of downstream applications, including reasoning, decision making, question answering, to name a few. However, this is a challenging problem due to the inherent non-unique graph representation (graph with N nodes can have $N!$ equivalent adjacency matrices), complex node and edge structure (node set is not fixed and edges are not binary), large output spaces (for graph with N nodes the system may need to output up to N^2 edges to specify its structure), lack of efficient architectures specialized for graph-structured generation output and limited parallel training data.

The related problem of generating text from a given KG is generally more widely studied, with many suggested architectures and approaches.

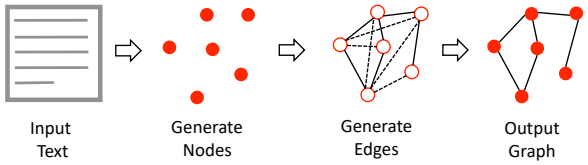

Figure 1: Grapher overview. For a given text input, in the first step we generate graph nodes, leveraging the representation power of pre-trained language models, fine-tuned on the task of entity extraction. In the second step, the graph edges are generated using the available entity information to construct the final graph.

Among the proposed methods, some of the current state-of-the-art systems that work on small or moderately-sized graphs, (Li et al., 2020; Ribeiro et al., 2020; Agarwal et al., 2020; Xie et al., 2022), usually formulate it as a simple sequence-to-sequence problem by representing the graph in a linearized form and fine-tune the pre-trained language models (PLMs), such as T5 (Raffel et al., 2020) or BART (Lewis et al., 2020), on the task of translating the sequence of triples to the corresponding textual description.

Nevertheless, KG generation remains a popular research area, receiving attention from many communities, including natural language processing (NLP), data mining, and machine learning. Recent success of the Transformer-based language models from the NLP community (Vaswani et al., 2017; Devlin et al., 2019; Raffel et al., 2020), pre-trained on large textual corpora, led to a series of works that attempted to exploit the vast amounts of learned linguistic knowledge for the downstream task of KG construction. Some of these approaches looked into a simpler problem of graph completion (Li et al., 2016; Yao et al., 2019; Malaviya et al., 2020). The drawback of these methods is that they are limited to the task of extending existing graphs by local neighborhood modifications and are not suitable for building the entire global graph structures. Alternatively, other works (Petroni et al.,

Findings of the Association for Computational Linguistics: EMNLP 2022, pages 1610 - 1622

2019; Roberts et al., 2020; Jiang et al., 2019; Shin et al., 2020; Li and Liang, 2021) proposed to query the pre-trained models to extract the learned factual and commonsense knowledge. The idea is to prompt the language model to predict the masked objects in cloze sentences describing the partially complete triples. Similarly as before, these methods are usually only suitable for local graph patching, lacking the ability to perceive the global graph structure.

Alternatively, there are a number of works that propose to generate the entire graph structure ground up. One example is GraphRNN from You et al. (2018), which models a graph as a sequence of additions of new nodes using node-level RNN and edges using another edge-level RNN. Although promising for our task of KG construction, the sequential and greedy nature of its generation can cause sub-optimal graph structures. CycleGT of (Guo et al., 2020b) is an unsupervised method for text-to-graph and graph-to-text generation, where the graph generation part relies on off-the-shelf entity extractor followed by a classifier to predict the relationships. The reliance on external NLP pipelines breaks the end-to-end continuity of system training, potentially leading to sub-optimal results. Similarly, (Dognin et al., 2020) proposed DualTKB employing unsupervised cycle loss to enable the graph-text translation in both directions. However, their method was applied only to single sentence-single triple generation, limiting applicability for larger graphs. Other approaches, such as BT5 from (Agarwal et al., 2020) proposed to utilize large pre-trained T5 model to generate KG in a linearized form, where the object-predicate-subject triples are concatenated together and the entire text-to-graph problem is viewed as sequence-to-sequence modeling. The potential issue with this approach is that the graph linearization is not unique and inefficient due to the repetition of graph components multiple times, leading to long sequences and increased complexity. (Lu et al., 2022) is another text-to-structure method, however it uses predefined schema (e.g., for entity or triplet extraction), while our method is schema-free and generalizes to any text form of nodes and edges. Finally, (Wang et al., 2020) proposed MaMa for KG construction, where entities and relationships are first matched using the attention weight matrices from the forward pass of the LM. Those are then mapped to the existing KG schema to generate the final graph.

The proposed system: Grapher Analyzing the shortcomings of the existing methods, in this work we propose to address them with a novel Knowledge Graph construction system which we call Grapher, presented schematically in Fig. 1. Given input text, the graph generation is split into two steps. In the first step, we leverage the representation power of pre-trained language models, e.g., T5 (Raffel et al., 2020), fine-tuned on the task of entity (graph nodes) extraction, while in the second stage the relationships (graph edges) are generated using the available entity information. There are three main properties of Grapher: **(i)** The use of state-of-the-art language models pre-trained on large textual corpora, used for node generation is key to the algorithm's performance as it lays out the foundation for the entire graph. The available parallel data for learning the text to graph translation is usually small, therefore training custom-built entity extraction architectures from scratch on this limited data is inferior to fine-tuning the already pretrained Transformer-based language models. **(ii)** The partitioning of graph construction process into two steps ensures efficiency that each node and edge is generated only once, which is in contrast to graph linearization approaches, e.g., (Agarwal et al., 2020) (Dognin et al., 2021), whose graph sequence representation is non-unique and can be inefficient. **(iii)** Finally, the entire system is end-to-end trainable, where the node and edge generation are optimized jointly, enabling efficient information transfer between the two modules, avoiding the need of any external NLP pipelines such as entity/relation extraction, co-reference resolution, etc. We evaluate the proposed Grapher on three datasets: the WebNLG+ 2020 Challenge (Ferreira et al., 2020) matching state-of-the-art performance for Text-to-RDF generation as well as on NYT (Riedel et al., 2010) and a recent large-scale TEKGEN (Agarwal et al., 2021) dataset showing strong results outperforming existing baselines.

2 Method

In this Section we cover the details of the proposed approach, first describing the functionality of the node generation in Section 2.1, followed by the edge generation in Section 2.3 and the discussion on edge imbalance problem in Section 2.4. In Fig. 2 we summarize all the architectural choices of the Grapher system. The branches marked with a red

cross denote the setups which in our earlier evaluations did not show advantage over the neighboring branch, e.g., the focal loss underperformed the sparse edge training for the text nodes combined with edge generation head. The branches with green check marks are the ones we select for further evaluation. The bold dark green check marks show two best performing systems across multiple experiments. In what follows, we now show the details of these choices.

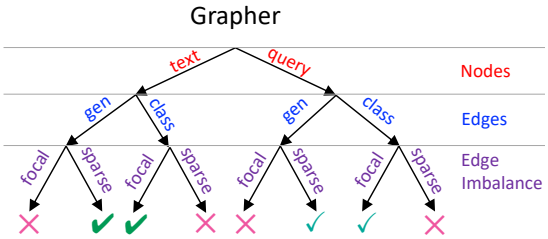

Figure 2: Grapher architectural choices. ✗ - setups that did not show advantage or did not perform well during preliminary evaluations, ✓ - selected for further evaluation , ✔ - best performing system

2.1 Node Generation: Text Nodes

Given text input, the objective of this module is to generate a set of unique nodes, which define the foundation of the graph. As we mentioned in Section 1, the node generation is key to the successful operation of Grapher, therefore for this task we use a pre-trained encoder-decoder language model (PLM), such as T5. Using a PLM, we can now formulate the node generation as a sequence-to-sequence problem, where the system is fine-tuned to translate textual input to a sequence of nodes, separated with special tokens, ⟨PAD⟩ NODE$_1$ ⟨NODE_SEP⟩ NODE$_2$ $\cdots$ ⟨/S⟩, where NODE$_i$ represents one or more words.

As seen in Fig. 3, in addition to node generation, this module supplies node features for the downstream task of edge generation. Since each node can have multiple associated words, we greedy-decode the generated string and utilize the separation tokens ⟨NODE_SEP⟩ to delineate the node boundaries and mean-pool the hidden states of the decoder's last layer. Note that in practice we fix upfront the number of generated nodes and fill the missing ones with a special ⟨NO_NODE⟩ token.

2.2 Node Generation: Query Nodes

One issue with the above approach is ignoring that the graph nodes are permutation invariant, since

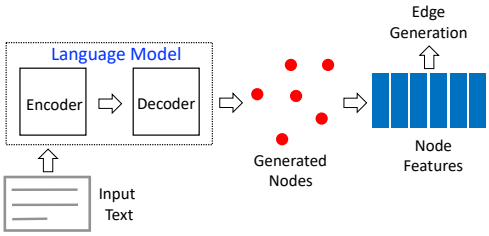

Figure 3: Node generation using traditional sequence-to-sequence paradigm based on T5 language model, where the input text is transformed into a sequence of text entities. The features corresponding to each entity (node) is extracted and sent to the edge generation module.

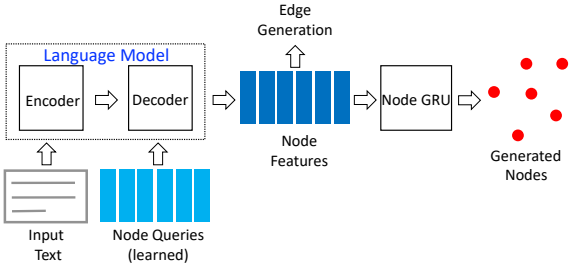

Figure 4: Node generation using learned query vectors. Here the input text and the query vectors (in the form of embedding matrix) is transformed into node features. Those are then decoded into graph nodes using node generation head (e.g, LSTM or GRU). The same features are also sent to the edge construction module.

any permutation of the given set of nodes should be treated equivalently. To address this limitation, we propose a second architecture, inspired by DETR (Carion et al., 2020). See Fig. 4 for an illustration.

Learnable Node Queries The decoder receives as input a set of learnable node queries, represented as an embedding matrix. We also disable causal masking, to ensure that the Transformer is able to attend to all the queries simultaneously. This is in contrast to the traditional encoder-decoder architecture that usually gets as an input embedding of the target sequence with the causal masking during training or the embedding of the self-generated sequence during inference. The output of the decoder can now be directly read-off as N (number of nodes) d-dimensional node features $F_n \in \mathbb{R}^{d \times N}$ and passed to a prediction head (LSTM or GRU) to be decoded into node logits $L_n \in \mathbb{R}^{S \times V \times N}$, where S is the generated node sequence length and V is the vocabulary size.

Permutation Matrix To avoid the system to memorize the particular target node order and enable permutation-invariance, the logits and features are

permuted as

$$L'_n(s) = L_n(s)P, \quad F'_n = F_nP, \qquad (1)$$

for $s = 1, \ldots, S$ and where $P \in \mathbb{R}^{N \times N}$ is a permutation matrix obtained using bipartite matching algorithm between the target and the greedy-decoded nodes. We used cross-entropy loss as the matching cost function. The permuted node features F'_n are now target-aligned and can be used in the edge generation stage.

2.3 Edge Generation

The generated set of node features from previous step is then used in this module for the edge generation. Fig. 5 shows a schematic description of this step. Given a pair of node features, a prediction head decides the existence (or not) of an edge between their respective nodes. One option is to use a head similar to the one in Section 2.2 (LSTM or GRU) to generate edges as a sequence of tokens. The other option is to use a classification head to predict the edges. The two choices have their own pros and cons and the selection depends on the application domain. The advantage of generation is the ability to construct *any* edge sequence, including ones unseen during training, at the risk of not matching the target edge token sequence exactly. On the other hand, if the set of possible relationships is fixed and known, the classification head is more efficient and accurate, however if the training has limited coverage of all possible edges, the system can misclassify during inference. We explore both options in Section 4.

Note that since in general KGs are represented as directed graphs, it is important to ensure the correct order (subject-object) between two nodes. For this, we propose to use a simple difference between the feature vectors: $F'_n(:, i) - F'_n(:, j)$ for the case when the node i is a parent of node j. We experimented with other options, including concatenation and adding position information but found the difference being the most effective, since the model learns that $F'_n(:, i) - F'_n(:, j)$ implies $i \rightarrow j$, while $F'_n(:, j) - F'_n(:, i)$ implies $j \rightarrow i$.

2.4 Imbalanced Edge Distribution

Observe that since we need to check the presence of edges between all pairs of nodes, we have to generate or predict up to N^2 edges, where N is the number of nodes. There are small savings that can be done by ignoring self-edges as well as ignoring edges when one of the generated nodes is the

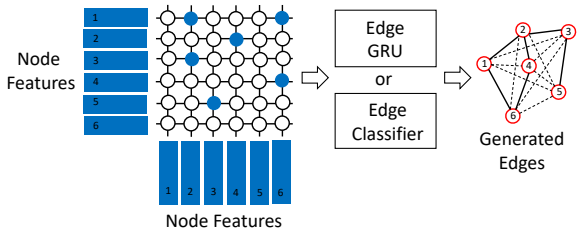

Figure 5: Edge construction, using generation (e.g., GRU) or a classifier head. Blue circles represent the features corresponding to the actual graph edges (solid lines) and the white circles are the features that are decoded into ⟨NO_EDGE⟩ (dashed line).

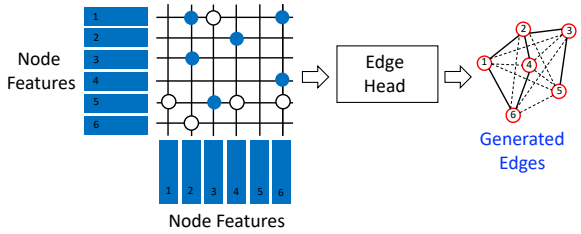

Figure 6: Edge generation with sparse adjacency matrix, using same decoder heads as in Fig. 5. Here while keeping all the actual edges, we remove most of the ⟨NO_EDGE⟩ tokens, leaving only a few. This setup is only used during training to improve the edge imbalance problem and speedup the training.

⟨NO_NODE⟩ token. When no edge is present between the two nodes, we denote this with a special token ⟨NO_EDGE⟩. Moreover, since in general the number of actual edges is small and ⟨NO_EDGE⟩ is large, the generation and classification task is imbalanced towards the ⟨NO_EDGE⟩ token/class. To remedy this, we propose two solutions: one is a modification of the cross-entropy loss, and the other is a change in the training paradigm.

Focal Loss Here we replace the traditional Cross-Entropy (CE) loss with Focal (F) loss (Lin et al., 2020), whose main idea is down-weight the CE loss for well-classified samples (⟨NO_EDGE⟩) and increase the CE loss for mis-classified ones, as illustrated below for a probability p corresponding to a single edge and t is a target class:

$$\text{CE} = -\log(p_t), \; \text{F} = -(1-p_t)^{\gamma} \log(p_t),$$

where $\gamma \geq 0$ is a weighting factor, such that $\gamma = 0$ makes both losses equivalent. The application of this loss to the classification head is straightforward while for the generation head we modify it by first accumulating predicted probabilities over the edge sequence length to get the equivalent of p_t and then apply the loss. In practice, we observed that Focal loss improved the accuracy for the clas-

Table 1: WebNLG dataset (Text-to-RDF)

	Train	Dev	Test
RDf triple sets	13,211	1,667	752
Texts	35,426	4,464	2,155

Table 2: Statistics of the TEKGEN dataset.

	Train	Dev	Test
Original	6,383,051	797,881	797,882
Processed	5,391,944	673,953	678,233

Table 3: Statistics of the NYT dataset.

	Train	Dev	Test
Normal	46,409	4,150	4,021

sification head, while for the generation head the performance did not change significantly.

Sparse Edges To address the edge imbalance problem another solution is to modify the training settings by sparsifying the adjacency matrix to remove most of the ⟨NO_EDGE⟩ edges as shown in Fig. 6, therefore re-balancing the classes artificially. Here, we keep all the actual edges but then leave only a few randomly selected ⟨NO_EDGE⟩ ones. Note that this modification is done only to improve efficiency of the training, during inference the system still needs to output all the edges, as in Fig. 5, since their true location is unknown. In practice, besides seeing 10-20% improvement in accuracy, we also observed about 10% faster training time when using sparse edges as compared to using full adjacency matrix.

3 Data

To evaluate Grapher's performance and compare it to the baselines, we use three datasets: two small-scale datasets: WebNLG+ 2020 (Ferreira et al., 2020) and NYT (Zeng et al., 2018), and a large-scale TEKGEN dataset from (Agarwal et al., 2021).

3.1 WebNLG+ 2020

The WebNLG+ corpus v3.0 is part of the 2020 WebNLG Challenge that offers two tasks: the generation of text from a set of RDF triples (subject-predicate-object), and the opposite task of semantic parsing for converting textual descriptions into sets of RDF triples. We preprocess the data to remove any underscores and surrounding quotes, in order to reduce noise in the data. Moreover, due to a mismatch of T5 vocabulary and the WebNLG dataset, some characters in WebNLG are not present in T5 vocabulary and ignored during tokenization. We normalize the data mapping the missing characters to the closest available, e.g., 'ø' is converted to 'o', or 'ã' is mapped to 'a'.

To prepare data for Grapher training, we split the triples into nodes (extracting subjects and objects) and edges (extracting predicates). The nodes are then either sequentially joined as ⟨PAD⟩ NODE_1 ⟨NODE_SEP⟩ NODE_2 ⟨/S⟩ for Text Nodes or passed separately as ⟨PAD⟩ NODE_1 ⟨/S⟩, ⟨PAD⟩ NODE_2 ⟨/S⟩ for Query Nodes, padding with ⟨NO_NODE⟩, if necessary. For edges, each element i, j of the adjacency matrix is filled with ⟨PAD⟩ $\text{EDGE}_{i,j}$ ⟨/S⟩ if there is an edge between NODE_i and NODE_j or with ⟨PAD⟩ NO_EDGE ⟨/S⟩ otherwise. In case sparse edges are used, we first sparsify the adjacency matrix, and then flatten it to a sequence of edges, similar as for the nodes. Finally, for the classification edge head we scan the training set and collect all the unique predicates to be the edge class list. There are 407 edge classes in our train split, including the ⟨NO_EDGE⟩ class.

3.2 TEKGEN

TEKGEN is a large-scale parallel text-graph dataset built by aligning Wikidata KG to Wikipedia text, and its statistics is shown in Table 2.

The data was preprocessed by filtering out triples containing more than 7 predicates, with triple components longer than 100 characters, and with corresponding textual descriptions longer than 200 characters. This was done to match the settings of the WebNLG data and to reduce the computational complexity of the scoring. The final statistics of the dataset is shown in the second row of Table 2. In total, the training set contains 1003 predicates/graph edges, which is more than twice larger than in the WebNLG dataset. Note that to match the evaluation to the baseline (Dognin et al., 2021), and to further manage the limited computational resources, we limit the Test split to 50K sentence-triples pairs.

3.3 NYT

As a third evaluation dataset, we selected the New York Times (NYT) corpus for our experiments, originally proposed by (Riedel et al., 2010), consisting of 1.18M sentences. We used an adapted version of the dataset pre-processed by (Zeng et al., 2018), referred as "normal", and contains the non-overlapping entities (i.e., head/tail pair has only

single edge connecting them), and 25 relation types (the smallest set as compared to WebNLG and TEKGEN). Table 3 shows the statistics of the dataset.

4 Experiments

In this Section we provide details about the model setups for evaluations, describe the scoring metrics, and present the results for both datasets.

4.1 Grapher Setup

For our base pre-trained language model we used T5 "large", for a total number of 770M parameters, from HuggingFace, Inc (Wolf et al., 2020) (see Appendix for the results using other model sizes). For Query Node generation we also defined the learnable query embedding matrix $M \in \mathbb{R}^{H \times N}$, where $H = 1024$ is the hidden size of T5 model, and $N = 8$ is the maximum possible number of nodes in a graph. The node generation head uses single-layer GRU decoder with $H_{\text{GRU}} = 1024$ followed by linear transformation projecting to the vocabulary of size $32,128$. The same GRU setup is used for the edge generation head, where we also set the maximum number of edges to be 7. Finally, for the edge classification head, we defined four fully-connected layers with ReLU non-linearities and dropouts with probability 0.5, projecting the output to the space of edge classes.

During training we fine-tuned all the model's parameters, using the AdamW optimizer with learning rate of 10^{-4}, and default values of $\beta = [0.9, 0.999]$ and weight decay of 10^{-2}. The batch size was set to 10 samples using a single NVIDIA A100 GPU for WebNLG and NYT training, while for TEKGEN training we employed distributed training over 10 A100 GPUs, thus making the effective batch size of 100. Under these settings, it takes approximately 3,500 steps to complete a training epoch for WebNLG, together with the validations done every 1,000 steps, we get a model that reaches its top performance in approximately 6-7 hours. For NYT, the epoch takes approximately 4,600 mini-batches, achieving top performance in about 15 epochs (24 hours). Finally, TEKGEN, each epoch takes approximately 54,000 steps, with the evaluations done every 1,000 steps we trained and validated the model for 150,000 iterations, taking approximately 14 days of compute time.

4.2 Baselines

To evaluate the performance of Grapher, for baselines we selected the top performing teams reported on the WebNLG 2020 Challenge Leaderboard, and briefly describe them here: **Amazon AI (Shanghai)** (Guo et al., 2020a) was the Challenge winner for Text-to-RDF task. They followed a simple heuristic-based approach that first does entity linking to match the entities present in the input text with the DBpedia ontology, and then query the DBpedia database to extract the relation between them. **BT5** (Agarwal et al., 2020) came in second place and used large pre-trained T5 model to generate KG in a linearized form, where the object-predicate-subject triples are concatenated together and the entire text-to-graph problem is viewed as a traditional sequence-to-sequence modeling. **CycleGT** (Guo et al., 2020b), third place contestant, followed an unsupervised method for text-to-graph and graph-to-text generation, where the KB construction part relies on off-the-shelf entity extractor to identify all the entities present in the input text, and a multi-label classifier to predict the relation between pairs of entities. **Stanford CoreNLP Open IE** (Manning et al., 2014): This is an unsupervised approach that was run on the input text part of the test set to extract the subjects, relations, and objects to produce the output triplets to give a baseline performance for the WebNLG 2020 Challenge. **ReGen** (Dognin et al., 2021): Recent work that leverages T5 pretrained language model and Reinforcement Learning (RL) for bidirectional text-to-graph and graph-to-text generation, which, similarly to Agarwal et al. (2020), also follows the linearized graph representation approach.

4.3 Evaluation Metrics

For scoring the generated graph, we used the evaluation scripts from WebNLG 2020 Challenge (Ferreira et al., 2020), which computes the Precision, Recall, and F1 scores for the output triples against the ground truth. In particular, since the order of generated and ground truth triples should not influence the result, the script searches for the optimal alignment between each candidate and the reference triple through all possible permutation of the hypothesis-reference pairs. Then, the metrics based on Named Entity Evaluation (Segura-Bedmar et al., 2013) were used to measure the Precision, Recall, and F1 score in four different ways. **Exact**: The candidate triple should match exactly the reference

Table 4: Evaluation results on the test set of the WebNLG+ 2020 dataset. The top four block-rows are the results taken from the WebNLG 2020 Challenge Leaderboard (Ferreira et al., 2020). The bottom part shows the results of our proposed Grapher system for several architectural choices, as discussed in Section 2. Bold font shows the best performing systems.

			M.	F1	Prec.	Rec.
Amazon AI			E	0.689	0.689	0.690
			P	0.696	0.696	0.698
			S	0.686	0.686	0.687
BT5			E	0.682	0.670	0.701
			P	0.713	0.700	0.736
			S	0.675	0.663	0.695
CycleGT			E	0.342	0.338	0.349
			P	0.360	0.355	0.372
			S	0.309	0.306	0.315
Stanford OIE			E	0.158	0.154	0.164
			P	0.200	0.194	0.211
			S	0.127	0.125	0.130
ReGen			E	**0.723**	**0.714**	**0.738**
			P	**0.767**	**0.755**	**0.788**
			S	**0.720**	**0.713**	**0.735**
Grapher	Query Nodes	Gen Edges	E	0.395	0.391	0.400
			P	0.325	0.318	0.337
			S	0.289	0.285	0.294
		Class Edges	E	0.466	0.463	0.469
			P	0.360	0.356	0.368
			S	0.347	0.345	0.351
	Text Nodes	Gen Edges	E	0.683	0.675	0.695
			P	0.713	0.702	0.730
			S	0.681	0.673	0.693
		Class Edges	E	**0.722**	**0.715**	**0.733**
			P	**0.750**	**0.741**	**0.765**
			S	**0.719**	**0.712**	**0.730**

triple, while the type (subject, predicate, object) is not important. **Partial**: The candidate triple should match at least partially with the reference triple, while the type (subject, predicate, object) is irrelevant. **Strict**: The candidate triple matches exactly the reference triple, and the element type (subject, predicate, object) should match exactly as well.

4.4 WebNLG Results

The main results for evaluating all the compared methods on WebNLG test set are presented in Table 4. As one can see, our Grapher system, based on Text Nodes and Class Edges, achieved on par top performance, as the ReGen (Dognin et al., 2021) model. Our system also uses the Focal loss to account for edge imbalance during training. We can also see that Grapher based on Text Nodes, where the T5-based model generates the nodes directly as a string, outperforms the alternative approach that generates the nodes through query vectors and permutes the features to get invariance to node ordering. A possible explanations is that the graphs at hand and the training data are both quite small. Therefore, the representational power of T5, pre-trained on textual corpora several orders of magnitude larger, can handle the entity extraction task much better. As we mentioned earlier, the ability to extract the nodes is very crucial to the overall success of the system, so if the query-based node generation constructs less reliable sets of nodes, the follow-up stage of edge generation will underperform as well.

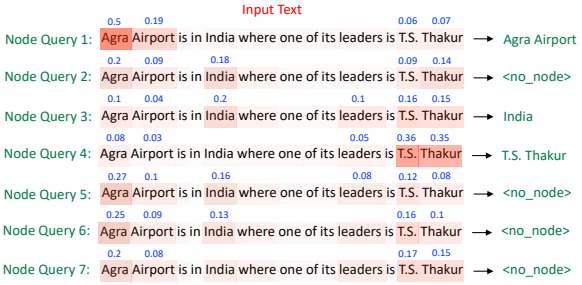

Figure 7: Visualization of the cross-attention weights in the T5 model between the node query embedding vectors and the embeddings of the input text.

Comparing the edge generation versus classification, we see that the former approach already brings up the system to the level of the top two leaderboard performers, while the edge classification adds extra accuracy and makes Grapher one of the leading system. This again might be due to a smaller training set, in which case GRU edge decoder underperforms, generating less accurate edges, while the classifier just needs to predict a single class to construct an edge, making it a better alternative in the low-data scenarios.

Finally, note that although the query-based node generation did not perform well in our evaluations, it is still informative to examine the behaviour of these vectors learned during the training. For this, we analyze the cross-attention weights in the T5 model between the node query vectors and the embeddings of the input text; the results are shown in Fig. 7. The ground truth nodes for this sentence are 'Agra Airport', 'India' and 'T.S. Thakur'. It can be seen that each query vector focuses on a set of words that can potentially become a node. For example, the first query vector emphasizes the words 'Agra', 'Airport', 'T.S.' and 'Thakur', but since the

Table 5: Evaluation results on the test set of TEKGEN dataset for different configurations of the Grapher system. The use of text-based nodes with generation edges performs the best.

			M.	F1	Prec.	Rec.
ReGen			E	0.623	0.610	0.647
Grapher	Query Nodes	Gen Edges	E	0.386	0.361	0.430
			P	0.438	0.405	0.496
			S	0.386	0.361	0.430
		Class Edges	E	0.361	0.338	0.401
			P	0.408	0.378	0.463
			S	0.360	0.337	0.401
	Text Nodes	Gen Edges	E	**0.707**	**0.693**	**0.730**
			P	**0.741**	**0.723**	**0.771**
			S	**0.706**	**0.692**	**0.729**
		Class Edges	E	0.700	0.686	0.722
			P	0.735	0.717	0.764
			S	0.700	0.685	0.721

weight on the first two words is higher, the resulting feature vector sent to the Node GRU module correctly decodes it as 'Agra Airport'. The same process happens for the third and forth query vectors. It is also interesting to see that the rest of the queries were also correctly decoded as ⟨NO_NODE⟩ token, even though they had high attention weights on some of the words (e.g., weight of 0.2 on 'Agra' and 0.18 on 'India' for the second query vector). One potential explanation is that since no causal mask is used when feeding query vectors to the decoder, T5 has an opportunity to exchange the information between all of the query vectors across all the layers and heads. Thus, once the found nodes are assigned to specific vectors, the rest of them are suppressed and decoded into ⟨NO_NODE⟩, irrespective of the attention weights.

4.4.1 TEKGEN Results

The results on the test set of the TEKGEN dataset (Agarwal et al., 2021) are shown in Table 5. To compute the graph generation performance, we use the same scoring functions as in WebNLG 2020 Challenge (Ferreira et al., 2020). As in Table 4, in this experiment we observe a similar pattern in which the Grapher based on Text Nodes outperforms the query-based system. At the same time we see now that the GRU-based edge decoding performs better than the classification edge head. Recall that for the smaller-size WebNLG dataset the classification edge head performed better, while now on the larger-size TEKGEN dataset, the GRU edge generation is more accurate, outperforming the simpler classification edge head. Also, our Grapher model now outperforms the ReGen baseline from (Dognin et al., 2021), which is based on the linearization technique to represent the graph, showing advantage of the proposed multi-stage generation approach.

Table 6: Evaluation results on the test set of NYT dataset for different configurations of the Grapher system. Text-based nodes with generation edges performs the best.

			M.	F1	Prec.	Rec.
T5 + Linearized Graph			E	0.832	0.831	0.834
			P	0.834	0.832	0.837
			S	0.824	0.822	0.826
Grapher	Text Nodes	Gen Edges	E	**0.918**	**0.917**	**0.920**
			P	**0.919**	**0.918**	**0.921**
			S	**0.913**	**0.911**	**0.914**
		Class Edges	E	0.870	0.867	0.872
			P	0.871	0.869	0.874
			S	0.860	0.858	0.862

4.4.2 NYT Results

Finally, Table 6 shows the results on NYT dataset. Similar as for the TEKGEN, Grapher based on text nodes and generation edges performs the best, outperforming the other architectural choices and the baseline (note that this baseline is our own implementation similar to (Dognin et al., 2021) and (Agarwal et al., 2020), which uses T5 pre-trained language model on the linearized graph representation). Comparing with the results from Tables 4 and 5, we can see that for smaller datasets, the classification head has a clear advantage, while as more training data becomes available, the GRU edge decoder becomes more accurate, outperforming the classifier edge head.

5 Conclusion

In this work, we proposed Grapher, a novel multi-stage KG generation system, that separates the overall graph generation into two steps. In the first step, the nodes are generated from the input text using a pretrained language model. The resulting node features are then used for edge generation to construct the output graph. We proposed several architectural choices for each stage. In particular, graph nodes can either be generated as a sequence of text tokens or as a set of query-based feature vectors decoded into tokens through generation head (e.g., GRU). Edges can be either generated by a GRU decoding head or selected by a classification head. We also addressed the problem of skewed edge

distribution, where the token/class corresponding to the missing edge is over-represented, leading to inefficient training. For this, we proposed to use of either the focal loss, or the sparse adjacency matrix. The experimental evaluations showed that Grapher matched state-of-the-art performance on smaller WebNLG dataset, and showed strong overall performance, outperforming existing baselines, on NYT and TEKGEN datasets, serving as a viable alternative to the existing baselines.

Limitations

There are several limitations of this work that need to be addressed in the future work. The first is the computational complexity of edge generation, which is quadratic in the number of edges, and this sets the limit on the sizes of the graphs that the systems can process. Moreover, since the nodes are generated using transformer-based models, which have quadratic complexity of the attention mechanism, there is a limit on the size of the input text the system can handle. Therefore, the current algorithm is suitable for small or medium size graphs and text passages. The extension to large scale is important and will be a part of the future effort. Moreover, the current setup was applied only to English domain datasets, which is a limitation, given that there is a benefit of multi- and cross-lingual training of language systems as ours. Finally, although not being our objective, the current model is designed to handle only the direction from text to knowledge graph, and the reverse direction has not been explored yet but can be a part of the future investigation.

References

Oshin Agarwal, Heming Ge, Siamak Shakeri, and Rami Al-Rfou. 2021. Knowledge graph based synthetic corpus generation for knowledge-enhanced language model pre-training. In *Proceedings of the Association for Computational Linguistics*, pages 3554–3565.

Oshin Agarwal, Mihir Kale, Heming Ge, Siamak Shakeri, and Rami Al-Rfou. 2020. Machine translation aided bilingual data-to-text generation and semantic parsing. In *Proceedings of the International Workshop on Natural Language Generation from the Semantic Web*.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-end object detection with transformers. *ArXiv*, abs/2005.12872.

J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*.

Pierre Dognin, Igor Melnyk, Inkit Padhi, Cicero Nogueira dos Santos, and Payel Das. 2020. DualTKB: A Dual Learning Bridge between Text and Knowledge Base. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Pierre L. Dognin, Inkit Padhi, Igor Melnyk, and Payel Das. 2021. Regen: Reinforcement learning for text and knowledge base generation using pretrained language models.

Thiago Castro Ferreira, Claire Gardent, N. Ilinykh, C. Lee, Simon Mille, Diego Moussallem, and Anastasia Shimorina. 2020. The 2020 Bilingual, Bi-Directional WebNLG+ Shared Task: Overview and Evaluation Results (WebNLG+ 2020). In *International Workshop on Natural Language Generation from the Semantic Web*.

Qipeng Guo, Zhijing Jin, Ning Dai, Xipeng Qiu, Xiangyang Xue, David Wipf, and Zheng Zhang. 2020a. P2: A plan-and-pretrain approach for knowledge graph-to-text generation. In *Proceedings of the International Workshop on Natural Language Generation from the Semantic Web*.

Qipeng Guo, Zhijing Jin, Xipeng Qiu, W. Zhang, D. Wipf, and Zheng Zhang. 2020b. CycleGT: Unsupervised graph-to-text and text-to-graph generation via cycle training. *ArXiv*, abs/2006.04702.

Zhengbao Jiang, F. F. Xu, J. Araki, and Graham Neubig. 2019. How can we know what language models know? *Transactions of the Association for Computational Linguistics*, 8:423–438.

M. Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *ArXiv*, abs/1910.13461.

Xiang Li, Aynaz Taheri, Lifu Tu, and Kevin Gimpel. 2016. Commonsense Knowledge Base Completion. In *Proceedings of the Annual Meeting of the ACL*, pages 1445–1455.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *ArXiv*, abs/2101.00190.

Xintong Li, Aleksandre Maskharashvili, S. Stevens-Guille, and Michael White. 2020. Leveraging large pretrained models for WebNLG 2020. In *International Workshop on Natural Language Generation from the Semantic Web*.

Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2020. Focal loss for dense object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42:318–327.

Yaojie Lu, Qing Liu, Dai Dai, Xinyan Xiao, Hongyu Lin, Xianpei Han, Le Sun, and Hua Wu. 2022. Unified structure generation for universal information extraction. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5755–5772.

Chaitanya Malaviya, Chandra Bhagavatula, Antoine Bosselut, and Yejin Choi. 2020. Commonsense Knowledge Base Completion with Structural and Semantic Context. *The Association for the Advancement of Artificial Intelligence*.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the Association for Computational Linguistics*.

Fabio Petroni, Tim Rocktäschel, Patrick Lewis, A. Bakhtin, Yuxiang Wu, Alexander H. Miller, and S. Riedel. 2019. Language models as knowledge bases? In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer.

Leonardo F. R. Ribeiro, Martin Schmitt, H. Schutze, and Iryna Gurevych. 2020. Investigating pretrained language models for graph-to-text generation. *ArXiv*, abs/2007.08426.

Sebastian Riedel, Limin Yao, and Andrew McCallum. 2010. Modeling relations and their mentions without labeled text. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 148–163.

Adam Roberts, Colin Raffel, and Noam M. Shazeer. 2020. How much knowledge can you pack into the parameters of a language model? In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Isabel Segura-Bedmar, Paloma Martínez, and María Herrero-Zazo. 2013. SemEval-2013 Task 9 : Extraction of drug-drug interactions from biomedical texts (ddiextraction 2013). In *SemEval@NAACL-HLT*.

Taylor Shin, Yasaman Razeghi, IV RobertLLogan, Eric Wallace, and Sameer Singh. 2020. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *ArXiv*, abs/2010.15980.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*.

C. Wang, Xiao Liu, and D. Song. 2020. Language models are open knowledge graphs. *ArXiv*, abs/2010.11967.

Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45.

Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2022. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. *EMNLP*.

Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. KG-BERT: BERT for Knowledge Graph Completion. *arXiv preprint arXiv:1909.03193*.

Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and J. Leskovec. 2018. GraphRNN: Generating realistic graphs with deep auto-regressive models. In *ICML*.

Xiangrong Zeng, Daojian Zeng, Shizhu He, Kang Liu, and Jun Zhao. 2018. Extracting relational facts by an end-to-end neural model with copy mechanism. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 506–514.

A Appendix

In Tables 7 and 8 we present the results of the best performing Grapher configurations, which uses Text Nodes with either Class Edges or Gen Edges respectively, with multiple random initializations to examine the results variability on WebNLG and NYT test set. As can be seen, the scores averaged across 5 runs (with different random initializations) show low standard deviation, further validating Grapher's stable performance.

Table 7: Mean and standard deviation for the results of 5 randomly initialized runs of the best Grapher configuration which uses Text Nodes and Class Edges on WebNLG test set.

Match	F1	Precision	Recall
Exact	$0.720_{\pm 0.05}$	$0.711_{\pm 0.05}$	$0.729_{\pm 0.06}$
Partial	$0.744_{\pm 0.04}$	$0.737_{\pm 0.03}$	$0.763_{\pm 0.03}$
Strict	$0.716_{\pm 0.05}$	$0.709_{\pm 0.05}$	$0.724_{\pm 0.05}$

Table 8: Mean and standard deviation for the results of 5 randomly initialized runs of the best Grapher configuration which uses Text Nodes and Gen Edges on NYT test set.

Match	F1	Precision	Recall
Exact	$0.908_{\pm 0.06}$	$0.909_{\pm 0.05}$	$0.910_{\pm 0.04}$
Partial	$0.910_{\pm 0.03}$	$0.910_{\pm 0.04}$	$0.908_{\pm 0.05}$
Strict	$0.905_{\pm 0.04}$	$0.903_{\pm 0.04}$	$0.904_{\pm 0.05}$

In Tables 9 and 10 we also present additional experiments by varying the T5 model size. In particular, in addition to the T5-large model, containing 770M parameters, used in the main paper, we also considered T5-base (220M parameters) and T5-small (60M parameters). It can be seen, that in general the performance drops as the model size decreases. However, for NYT dataset, the model architecture that uses Text Nodes and Class Edges, T5-small actually outperforms T5-base. At the same time, for Gen Edges all three model choices performed very similar with minor drop in performance as the size decreases.

Table 9: WebNLG

			Large	Base	Small
Grapher	Text Nodes	Gen Edges	**0.683**	0.660	0.596
		Class Edges	**0.722**	0.693	0.631

Table 10: NYT

			Large	Base	Small
Grapher	Text Nodes	Gen Edges	**0.912**	0.907	0.897
		Class Edges	**0.870**	0.812	0.846

In Fig. 8 we present some examples of the generated graphs (right column) and their associated ground truths (left column) for WebNLG dataset. In Fig. 9 similar results are given for TEKGEN dataset. Both examples show that the trained Grapher system sometimes can generated more detailed and accurate graphs corresponding to the input text as compared to the ground truth (e.g., first three examples in Fig. 9, where it adds extra edges for genre, occupation and birth date). Also, the use of T5 model for node extraction shows that the model can include information in the generated nodes that is not present in the input text (e.g., third example in Fig. 8, which included 'inhabitants per square kilometre', possibly from T5's original pretraining on large textual corpora.)

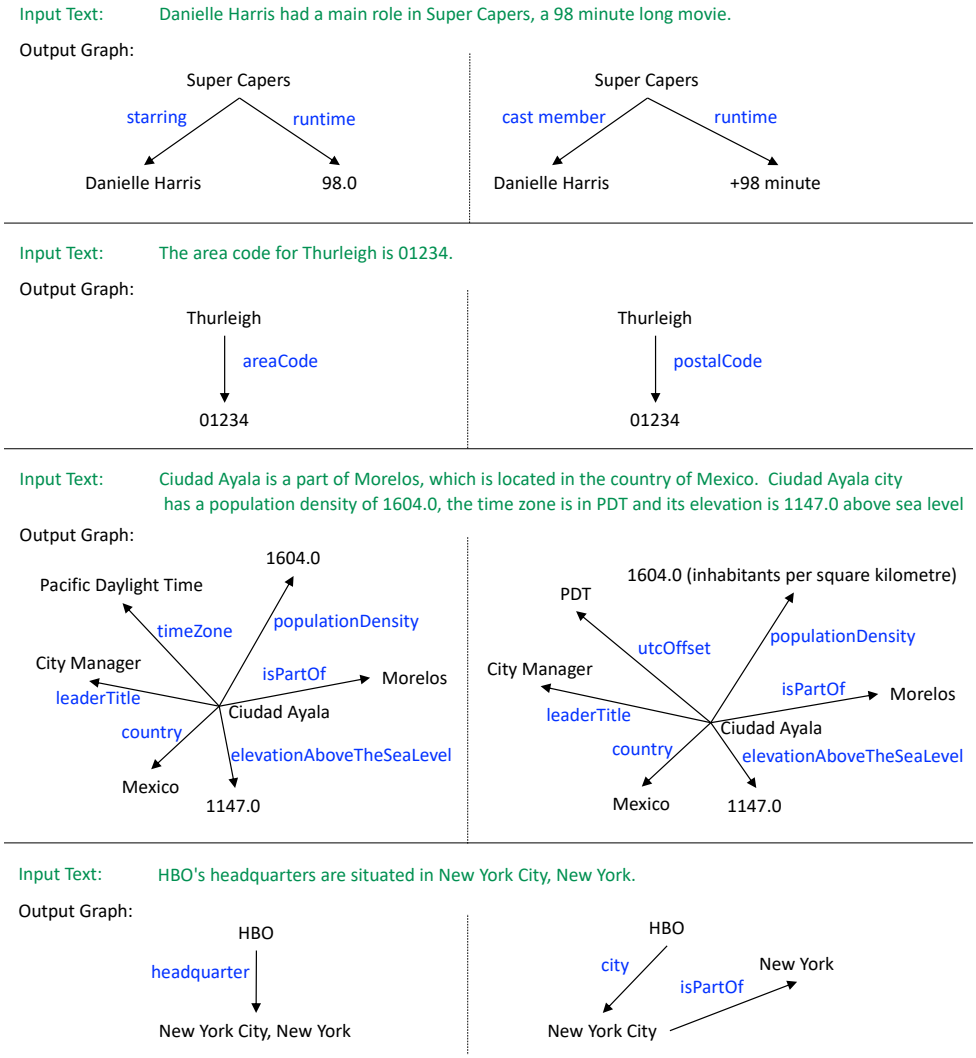

Figure 8: Examples of some of the notable generated (right column) and the ground truth graphs (left column) for WebNLG dataset.

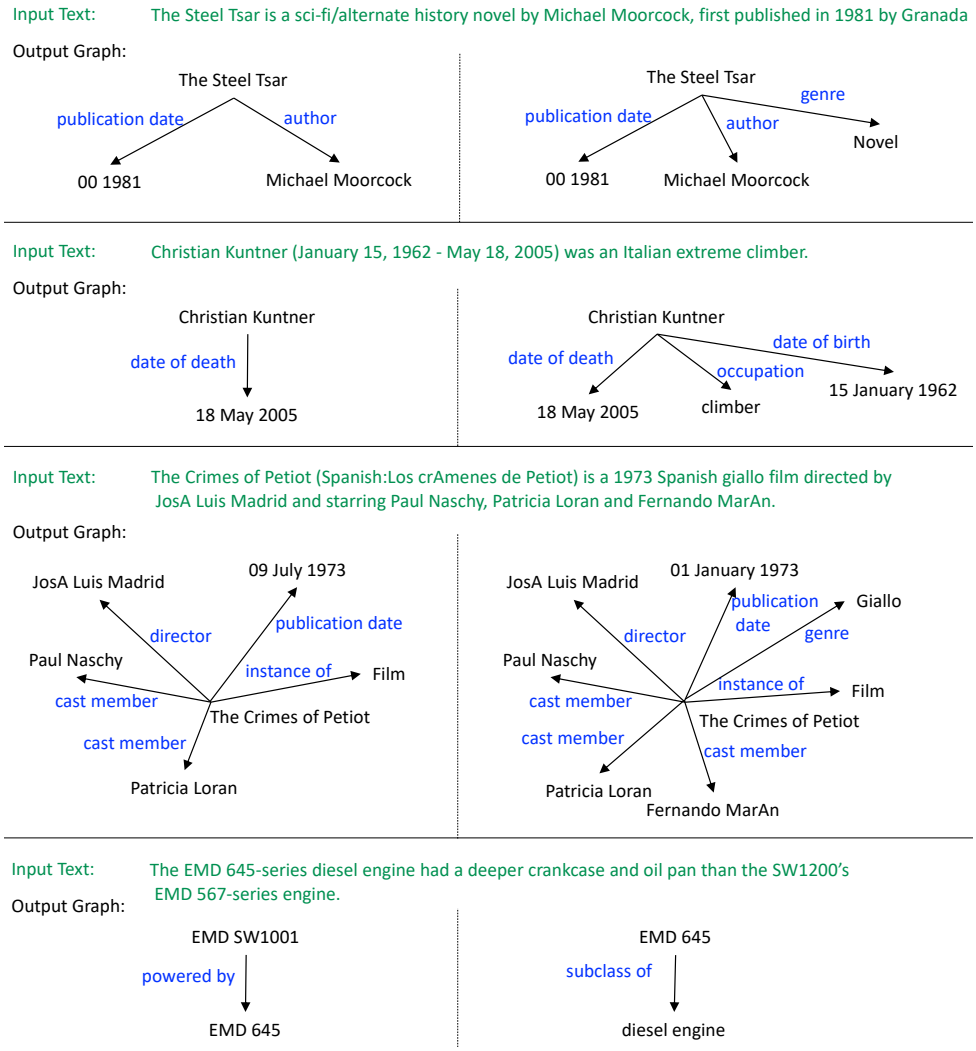

Figure 9: Examples of some of the notable generated (right column) and the ground truth graphs (left column) for TEKGEN dataset.