Grapher: Multi-Stage Knowledge Graph Construction using Pretrained Language Models

Igor Melnyk
IBM Research
igor.melnyk@ibm.com

Pierre Dognin
IBM Research
pdognin@us.ibm.com

Payel Das
IBM Research
daspa@us.ibm.com

Abstract

In this work we address the problem of Knowledge Graph (KG) construction from text, proposing a novel end-to-end multi-stage Grapher system, that separates the overall generation 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 textual descriptions. For each stage we proposed several architectural choices that can be used depending on the available training resources. We evaluated the Grapher on a recent WebNLG 2020 Challenge dataset, achieving competitive results on text-to-RDF generation task, as well as on a recent large-scale TEKGEN dataset, showing strong overall performance. We believe that the proposed Grapher 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. Among the proposed methods, some of the current state-of-the-art systems that work on small or moderately-sized graphs, [16, 24, 2], 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 [22] or BART [13], 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. Traditionally, KG construction has been addressed by data mining community using unsupervised Information Extraction (IE) systems [20] such as Stanford Open IE [3], Open IE 5.1 [26, 27], NELL [5], or YAGO [23], among many others. These systems are usually based on hand-crafted heuristics and rules to extract the subject-relationship-object triples from the sentences and therefore can easily scale to massive open-domain corpora. However, being the generic large-scale KG construction tools, the extracted entities or predicates often lack in precision and quality.

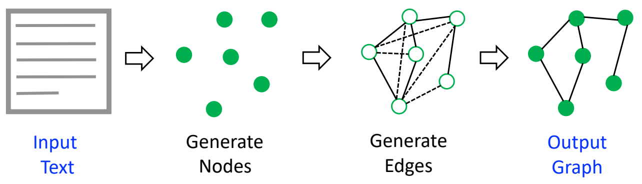

Figure 1: Overview of the proposed Grapher approach. Given text input, the graph generation is split into two steps. In the first step, we leverage the representation power of pre-trained language models, fine-tuned on the task of entity (graph nodes) extraction, while in the second stage the relationships (graph edges) are generated using the available entity information to construct the final graph.

Recent success of the Transformer-based language models from the NLP community [30, 6, 22], 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 [14, 33, 18], where, given a partial triple, usually missing one of the entities, the objective is to complete the triple by generating the missing entry or by ranking the given set of candidates. 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 [21, 25, 12, 29, 15] 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 [34], 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 [11] 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, [7] 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 [2] 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. Finally, [31] 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 [22], 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 usually 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., [2], 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 two datasets: the WebNLG+ 2020 Challenge [9] achieving new state-of-the-art results for Text-to-RDF generation as well as on a recent larger-scale TEKGEN dataset [1] showing strong performance on their validation and test splits.

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 mark shows our best performing system across multiple experiments. In what follows, we now show the details of these choices.

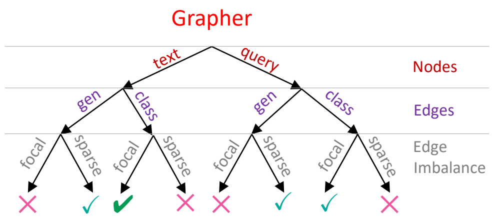

Figure 2: Grapher architectural choices. ✗ - setups that did not show advantage or did not perform well during preliminary evaluations, ✓ - selected for further evaluation , ✔ - best performing system

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, i.e., ⟨PAD⟩ NODE$_1$ ⟨NODE_SEP⟩ NODE$_2$ ⟨NODE_SEP⟩ NODE$_3$ ⟨/S⟩, where NODE$_i$ represents one or more words.

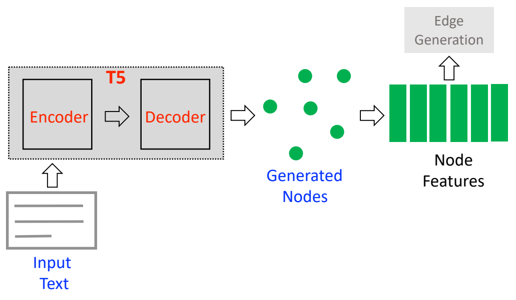

Figure 3: Node generation using traditional sequence-to-sequence paradigm based on T5 language model, where the input text is transformed into a sequence of text entities. The features corresponding to each entity (node) is extracted and sent to the edge generation module.

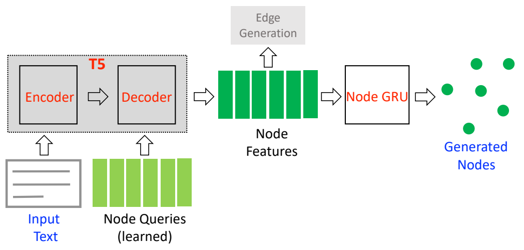

Figure 4: Node generation using learned query vectors. Here the input text and the query vectors (in the form of embedding matrix) is transformed into node features. Those are then decoded into graph nodes using node generation head (e.g, LSTM or GRU). The same features are also sent to the edge construction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

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 maximum 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 the fact that the graph nodes are permutation invariant, since any permutation of the given set of nodes should be treated equivalently. To address this limitation, we propose a second architecture, inspired by the DETR image object detection method of [4]. See Fig. 4 for an illustration, where we first learn node queries to get node features and then estimate the permutation to align with target node order.

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 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

$$L'_n(s) = L_n(s)P, \;\; F'_n = F_nP, \tag{1}$$

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 ⟨NO_NODE⟩ token. When no edge is present between the two nodes, we denote this with a special

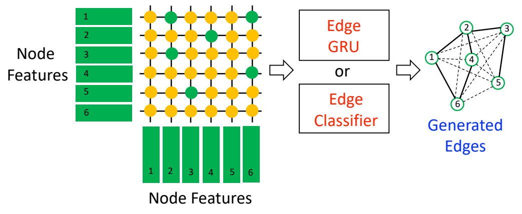

Figure 5: Edge construction, using generation (e.g., GRU) or a classifier head. Green circles represent the features corresponding to the actual graph edges (solid lines) and the orange circles are the features that decoded into ⟨NO_EDGE⟩ (dashed line).

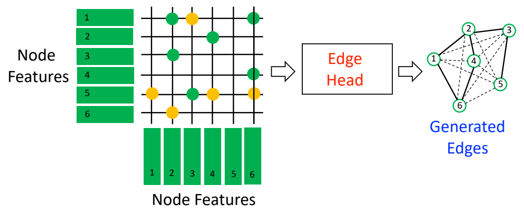

Figure 6: Edge generation with sparse adjacency matrix, using same decoder heads as in Fig. 5. Here while keeping all the actual edges, we remove most of the ⟨NO_EDGE⟩ tokens, leaving only a few. This setup is only used during training to improve the edge imbalance problem and speedup the training.

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 loss [17], whose main idea is down-weight the CE loss for well-classified samples (in our case ⟨NO_EDGE⟩) and increase the CE loss for misclassified ones, as illustrated below for a probability distribution p corresponding to a single edge and t is a target class:

$$\text{CE}(p,t) = -\log(p_t), \quad \text{FL}(p,t,\gamma) = -(1-p_t)^{\gamma}\log(p_t), \tag{2}$$

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 classification 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 the Grapher system performance and compare it to the baselines, we use two datasets: small-scale WebNLG+ 2020 [9] dataset, and a recent large-scale dataset TEKGEN from [1].

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. For our work, we evaluated the algorithm on the text-to-RDF task, whose statistics is shown in Table 1. Each triple set is associated with one or more lexicalizations, so when the triple set is assigned to all the lexicalizations, the total size of train, dev and test splits is shown in the second row of Table 1. The data consists of 16 DBpedia categories: 11 of which are present only in the train and dev splits, and 5 *unseen* categories which are part of the test split only.

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 vocabulary coverage between T5's tokenizer and the

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	5,598,909	699,964	699,851

WebNLG dataset, some characters in WebNLG are not present in T5 vocabulary and ignored during tokenization. We normalize the data to map those missing characters to the closest available, e.g., 'ø' is converted to 'o', or 'ã' is converted to 'a'.

Since WebNLG is a fairly small dataset, we additionally augment it with extra text-RDF pairs from the larger TEKGEN dataset (described later in Section 3.2). Since WebNLG does not use DBpedia schema, while TEKGEN is based on Wikidata KG, we do an approximate mapping of Wikidata predicates into the DBpedia ones. We only match the predicates in the training split. For example, a TEKGEN triple (SUBJECT, 'location of formation' OBJECT) is converted to a WebNLG triple as (SUBJECT, 'foundationPlace', OBJECT).

In total, we have added 17,855 pairs to the WebNLG dataset, making the total training set of size 53,281. 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 $\langle$PAD$\rangle$ NODE$_1$ $\langle$NODE_SEP$\rangle$ NODE$_2$ $\langle$/S$\rangle$ for Text Nodes or passed separately as $\langle$PAD$\rangle$ NODE$_1$ $\langle$/S$\rangle$, $\langle$PAD$\rangle$ NODE$_2$ $\langle$/S$\rangle$ for Query Nodes, padding with $\langle$NO_NODE$\rangle$, if necessary. For edges, each element i, j of the adjacency matrix is filled with $\langle$PAD$\rangle$ EDGE$_{i,j}$ $\langle$/S$\rangle$ if there is an edge between NODE$_i$ and NODE$_j$ or with $\langle$PAD$\rangle$ NO_EDGE $\langle$/S$\rangle$ 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 $\langle$NO_EDGE$\rangle$ class.

3.2 TEKGEN

TEKGEN is a recent large-scale parallel text-graph dataset built by aligning Wikidata KG to Wikipedia text, and its statistics is shown in the first row of Table 2. The main challenge in using TEKGEN is that the released data for triples is in the form (SUBJECT PREDICATE1 OBJECT1, PREDICATE2 OBJECT2, ...) separated by comas, which creates ambiguities during parsing as to what part of string belongs which component of the graph, since the predicates can be composed of multiple words (sometimes also separated by commas), where some of those words may also appear in the subject or the object.

Nevertheless, using the available list of Wikidata predicates, and a set of hand-crafted heuristics we were able to parse the data, additionally 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 approximate settings of the WebNLG data and to reduce the computational complexity of the scoring functions. The final statistics of the dataset is shown in the second row of Table 2. Note, that to further manage the limited computational resources, we only evaluated the results on half of the Dev and Test splits.

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 [32]. 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 of size 407.

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 while using a single NVIDIA A100 GPU for WebNLG training, while for TEKGEN training we employed distributed training over 10 A100 GPUs, thus making the effective batch size of 100. Under this settings, it takes approximately 5,300 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 TEKGEN, each epoch takes approximately 56,000 steps, with the evaluations done every 1,000 steps we trained and validated the model for 25,000 iterations, taking approximately 6 days of compute time.

4.2 Baselines

To evaluate the performance of Grapher, we use for baselines the top performing teams reported on the WebNLG 2020 Challenge Leaderboard, and briefly describe them as follows: **Amazon AI (Shanghai)** [10] 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** [2] 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** [11], 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** [19]: 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** [8]: This work leverages T5 pretrained langugage model and Reinforcement Learning (RL) for bidirectional text-to-graph and graph-to-text generation, which, similarly to [2], 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 [9], 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 [28] were used to measure the Precision, Recall, and F1 score in four different ways. **Exact**: The candidate triple should match exactly the reference 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 should match 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 3. As one can see, our Grapher system, based on Text Nodes followed by Class Edges, achieved the second best performance, closely following ReGen [8]. This 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

Table 3: 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 [9]. The bottom part shows the results of our proposed Grapher system for several architectural choices, as discussed in Section 2. Bold black and blue shows the best and second best performance, respectively.

			Match	F1	Precision	Recall
Amazon AI (Shanghai) [10]			Exact	0.689	0.689	0.690
			Partial	0.696	0.696	0.698
			Strict	0.686	0.686	0.687
BT5 [2]			Exact	0.682	0.670	0.701
			Partial	0.713	0.700	0.736
			Strict	0.675	0.663	0.695
CycleGT [11]			Exact	0.342	0.338	0.349
			Partial	0.360	0.355	0.372
			Strict	0.309	0.306	0.315
Stanford Open IE [19]			Exact	0.158	0.154	0.164
			Partial	0.200	0.194	0.211
			Strict	0.127	0.125	0.130
ReGen [8]			Exact	**0.723**	**0.714**	**0.738**
			Partial	**0.767**	**0.755**	**0.788**
			Strict	**0.720**	**0.713**	**0.735**
Grapher (Ours)	Query Nodes	Gen Edges	Exact	0.395	0.391	0.400
			Partial	0.325	0.318	0.337
			Strict	0.289	0.285	0.294
		Class Edges	Exact	0.466	0.463	0.469
			Partial	0.360	0.356	0.368
			Strict	0.347	0.345	0.351
	Text Nodes	Gen Edges	Exact	0.641	0.635	0.651
			Partial	0.672	0.664	0.687
			Strict	0.638	0.632	0.647
		Class Edges	Exact	**0.709**	**0.702**	**0.720**
			Partial	**0.735**	**0.725**	**0.750**
			Strict	**0.706**	**0.700**	**0.717**

node generation constructs less reliable sets of nodes, the follow-up stage of edge generation will underperform as well.

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 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.

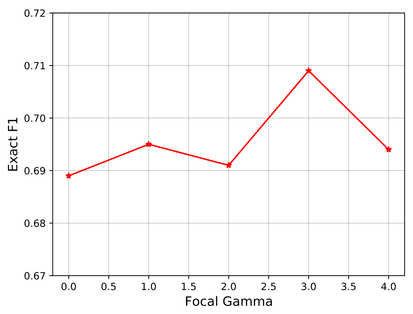

Figure 7: Impact of Focal parameter γ defined in equation (2) on Grapher performance, as measured by the F1 score with exact matching.

In Table 4 we present the results of the best performing Grapher configuration, which uses Text Nodes and Class Edges, with multiple random initializations to examine the results variability. As can be seen, the scores averaged across 3 runs (with different random initializations) show low standard deviation with the mean, still outperforming other baseline systems, further validating Grapher's good performance.

For the Edge Imbalance problem, we proposed two solutions: using Focal loss in place of Cross-Entropy loss, or using the sparse adjacency matrix during training. Since our best Grapher system

Table 4: Mean and standard deviation for the results of 3 randomly initialized runs of the best Grapher configuration which uses Text Nodes and Class Edges. The computed scores still make our method the top performer.

	Match	F1	Precision	Recall
Grapher	Exact	0.702 ± 0.05	0.695 ± 0.05	0.712 ± 0.06
	Partial	0.730 ± 0.04	0.721 ± 0.03	0.745 ± 0.03
	Strict	0.700 ± 0.05	0.693 ± 0.05	0.710 ± 0.05

uses Class Edges, it relies on Focal loss to address the above issue. In Fig. 7 we show the dependency of the F1 score (under the exact matching) on the Focal parameter $\gamma \geq 0$, defined in equation (2). Recall that γ reduces the relative loss for well-classified examples, while puts more emphasis on hard, mis-classified examples. We see that for our settings the performance is sensitive to the choice of γ, with $\gamma = 3$ achieving better results.

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. 8. 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 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 $\langle$NO_NODE$\rangle$ 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 $\langle$NO_NODE$\rangle$, irrespective of the attention weights.

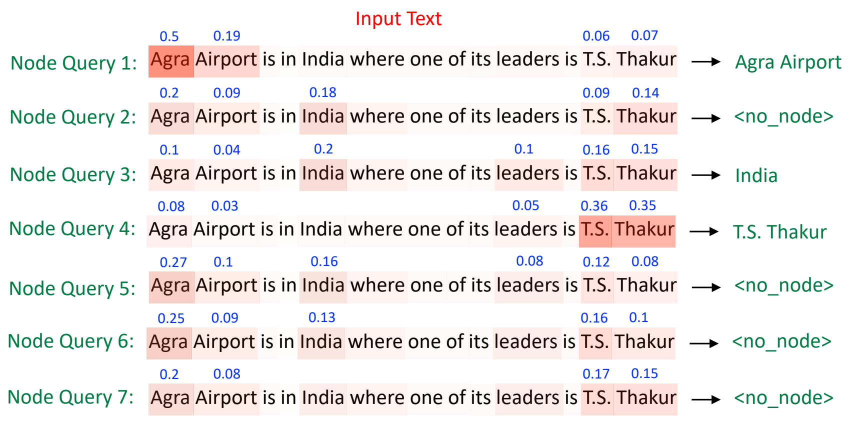
Input Text
Node Query 1: 0.5 0.19 0.06 0.07 Agra Airport is in India where one of its leaders is T.S. Thakur → Agra Airport
Node Query 2: 0.2 0.09 0.18 0.09 0.14 Agra Airport is in India where one of its leaders is T.S. Thakur → <no_node>
Node Query 3: 0.1 0.04 0.2 0.1 0.16 0.15 Agra Airport is in India where one of its leaders is T.S. Thakur → India
Node Query 4: 0.08 0.03 0.05 0.36 0.35 Agra Airport is in India where one of its leaders is T.S. Thakur → T.S. Thakur
Node Query 5: 0.27 0.1 0.16 0.08 0.12 0.08 Agra Airport is in India where one of its leaders is T.S. Thakur → <no_node>
Node Query 6: 0.25 0.09 0.13 0.16 0.1 Agra Airport is in India where one of its leaders is T.S. Thakur → <no_node>
Node Query 7: 0.2 0.08 0.17 0.15 Agra Airport is in India where one of its leaders is T.S. Thakur → <no_node>

Figure 8: Visualization of the cross-attention weights in the T5 model between the node query embedding vectors and the embeddings of the input text.

4.4.1 TEKGEN Results

The results on the test set (using only half of it to reduce the computational cost) of the TEKGEN dataset [1] are shown in Table 5. For computing the graph generation performance, we use the same scoring functions as in WebNLG 2020 Challenge [9]. As in Table 3, 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 similarly or 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, matching or outperforming the simpler classification edge head. As can be seen,

Table 5: Evaluation results on the test set of TEKGEN dataset for different configurations of the Grapher system. The use of text-based nodes and either classification edges or generation edges performs the best. Bold black and blue shows the best and second best performance, respectively.

			Match	**F1**	**Precision**	**Recall**
ReGen [8]			Exact	**0.623**	**0.610**	**0.647**
Grapher (Ours)	Query Nodes	Gen Edges	Exact	0.386	0.361	0.430
			Partial	0.438	0.405	0.496
			Strict	0.386	0.361	0.430
		Class Edges	Exact	0.361	0.338	0.401
			Partial	0.408	0.378	0.463
			Strict	0.360	0.337	0.401
	Text Nodes	Gen Edges	Exact	**0.641**	**0.626**	**0.666**
			Partial	**0.681**	**0.661**	**0.715**
			Strict	**0.640**	**0.625**	**0.665**
		Class Edges	Exact	**0.641**	**0.626**	**0.666**
			Partial	**0.681**	**0.661**	**0.715**
			Strict	**0.640**	**0.626**	**0.665**

our Grapher model now outperforms the ReGen baseline from [8], which utilizes the linearization technique to represent the graph.

Fig. 9 also confirms the importance of large datasets for Grapher, where it shows the validation F1 score across training iterations using the classification edge head (red line) or the generation edge head (blue line). During earlier training, the classification head has a clear advantage, while as the model sees more and more data, both choices converge, with the GRU decoder matching or even slightly outperforming the classifier edge head. From the plot we can also see that both models are still on the trajectory to improve their performance since the training has not gone through a single epoch, although the performance gain started to saturate, suggesting a limited improvement for the additional training.

5 Conclusion

In this work, we addressed the problem of Knowledge Graph construction from text, for which we proposed a novel multi-stage system named Grapher. The proposed system 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 in the second step of edge generation to construct the output graph. We proposed several architectural choices for each of the stages. In particular, all the 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 through a GRU-type decoding head or picked from a predefined set of edges through 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. To remedy this, we proposed use of either the focal loss (in place of the traditional cross-entropy loss), or the sparse adjacency matrix to make the training faster and more efficient. The experimental evaluations showed that the Grapher system showed strong overall performance for the text-to-RDF task on smaller WebNLG dataset, as well as for the graph generation on the recent larger-scale TEKGEN dataset, serving as a viable alternative to the existing baselines.

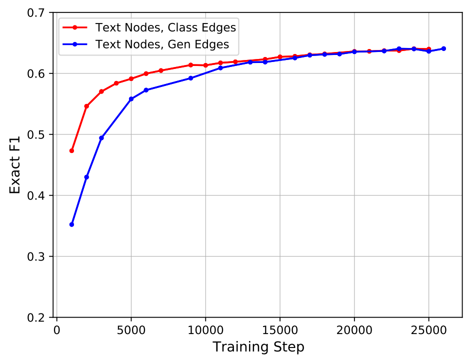

Figure 9: Validation F1 score (under exact matching) across training iterations on TEKGEN dataset for Grapher configurations using text-based nodes and either classification edges (red line) or generation edges (blue line).

References

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

[2] Oshin Agarwal, Mihir Kale, Heming Ge, Siamak Shakeri, and Rami Al-Rfou. 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*, 2020.

[3] Gabor Angeli, M. Johnson, and Christopher D. Manning. Leveraging linguistic structure for open domain information extraction. In *Proceedings of the Association for Computational Linguistics*, 2015.

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

[5] Andrew Carlson, J. Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka, and Tom Michael Mitchell. Toward an architecture for never-ending language learning. In *The Association for the Advancement of Artificial Intelligence*, 2010.

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

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

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

[9] Thiago Castro Ferreira, Claire Gardent, N. Ilinykh, C. Lee, Simon Mille, Diego Moussallem, and Anastasia Shimorina. 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*, 2020.

[10] Qipeng Guo, Zhijing Jin, Ning Dai, Xipeng Qiu, Xiangyang Xue, David Wipf, and Zheng Zhang. 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*, 2020.

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

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

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

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

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

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

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

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

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

[20] Mausam. Open information extraction systems and downstream applications. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, 2016.

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

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

[23] Thomas Rebele, Fabian M. Suchanek, Johannes Hoffart, J. Biega, Erdal Kuzey, and G. Weikum. YAGO: A multilingual knowledge base from wikipedia, wordnet, and geonames. In *International Semantic Web Conference*, 2016.

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

[25] Adam Roberts, Colin Raffel, and Noam M. Shazeer. 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*, 2020.

[26] Swarnadeep Saha and Mausam. Open information extraction from conjunctive sentences. In *Proceedings of the International Conference on Computational Linguistics*, 2018.

[27] Swarnadeep Saha, Harinder Pal, and Mausam. Bootstrapping for numerical Open IE. In *Proceedings of the Association for Computational Linguistics*, 2017.

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

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

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

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

[32] Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, et al. 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, 2020.

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

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