EaT-PIM: Substituting Entities in Procedural Instructions Using Flow Graphs and Embeddings

Sola S. Shirai[1(✉)] and HyeongSik Kim[2]

[1] Rensselaer Polytechnic Institute, Troy, NY 12180, USA
shiras2@rpi.edu
[2] Robert Bosch LLC, Sunnyvale, CA, USA
Hyeongsik.Kim@us.bosch.com

Abstract. When cooking, it can sometimes be desirable to substitute ingredients for purposes such as avoiding allergens, replacing a missing ingredient, or exploring new flavors. More generally, the problem of substituting entities used in procedural instructions is challenging as it requires an understanding of how entities and actions in the instructions interact to produce the final result. To support the task of automatically identifying viable substitutions, we introduce a methodology to (1) parse instructions, using NLP tools and domain-specific ontologies, to generate flow graph representations, (2) train a novel embedding model which captures flow and interaction of entities in each step of the instructions, and (3) utilize the embeddings to identify plausible substitutions. Our embedding strategy aggregates nodes and dynamically computes intermediate results within the flow graphs, which requires learning embeddings for fewer nodes than typical graph embedding models. Our rule-based flow graph generation method shows comparable performance to machine learning-based work, while our embedding model outperforms baselines on a link-prediction task for ingredients in recipes.

Keywords: Procedural instructions · Cooking recipes · Information extraction · Ingredient substitution · Knowledge graph embedding

1 Introduction

Procedural instructions are a valuable source of information which provide descriptions of how to carry out a task or achieve some goal. Such instructions are typically presented in a stepwise fashion, breaking down the overarching task

S. S. Shirai—Part of this work was done while the author was an intern at Robert Bosch LLC.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-19433-7_10.

U. Sattler et al. (Eds.): ISWC 2022, LNCS 13489, pp. 161–178, 2022.
https://doi.org/10.1007/978-3-031-19433-7_10

into a series of individual steps. A prime example of this is a cooking recipe, which specifies a set of ingredients along with a number of steps describing how to combine and modify those ingredients to form the final dish.

When performing tasks that are described by such instructions, it is possible to modify the instructions to complete the task in a slightly different way while producing similar results. In cooking, this can be observed when people substitute ingredients in the recipe – many ingredients exist that can be replaced and result in a dish that is "close enough" to the original. However, it can be difficult to determine which modifications of the instructions are valid because it requires an understanding of the entities involved with the instructions, the actions taking place, and the outcomes produced by different actions.

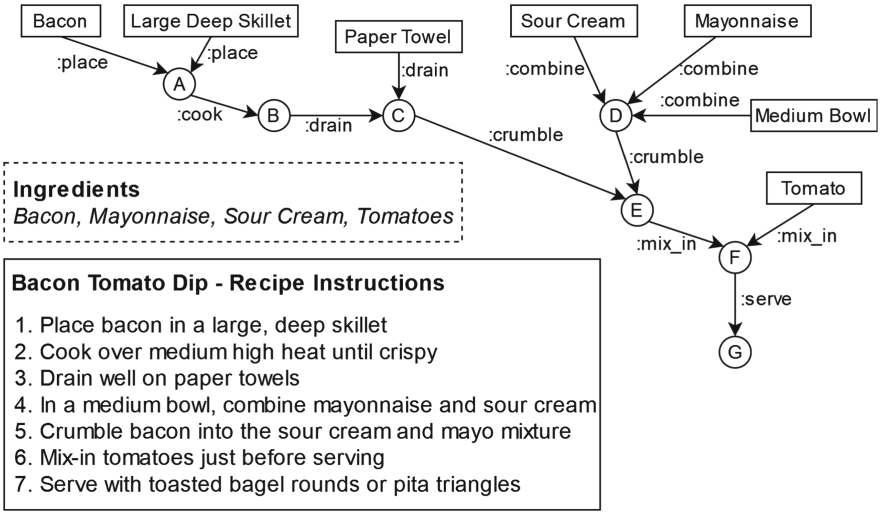

Fig. 1. A running example recipe and its flow graph. Intermediate nodes are labeled as A, B, etc. for convenience.

Gaining a comprehensive understanding about the entities and actions in procedural instructions presents a major challenge. Instructions often are not well structured or specific, as they rely on common sense. For example, given the instructions "(1) Place bacon in a skillet (2) Cook over medium heat", we infer that the instructions are telling us to cook the bacon that we just placed in the skillet. Correctly parsing these steps might also involve background knowledge, such as alternative names for similar entities (e.g., *pan* and *skillet*). Furthermore, steps are not necessarily completed sequentially, which requires us to identify branching instructions and co-references of similar entities from earlier steps.

One method that can help provide the structure necessary to represent this information and identify viable substitutions is to form a *flow graph* of the instructions. A flow graph can represent the instructions as a rooted, directed acyclic graph, with the root node representing the final result of the instructions

(e.g., the dish produced by a recipe), leaf nodes representing the entities (e.g., the ingredients and equipment), and edges capturing the actions taking place to produce intermediate results (e.g., mixing flour and water to form a batter). Representing the procedural instructions in this form can then be utilized to further identify which modifications can be made to the instructions.

A running example recipe is illustrated in Fig. 1. We can see several steps that specify how to use the ingredients, as well as equipment such as the skillet and bowl, to make the recipe. The recipe's corresponding flow graph captures these ingredient and equipment entities as leaf nodes, and their usages – i.e., verbs such as "cook" and "drain" – are captured as edges.

In order to form such flow graphs from procedural text, it can also be beneficial to incorporate domain-specific information sources. For example, ontologies can provide authoritative knowledge about entities that has been manually curated by domain experts. This knowledge in turn can inform the information extraction process and augment the resulting flow graph.

In this paper, we present the **EaT-PIM** (Embedding and Transforming Procedural Instructions for Modification) methodology to extract information from domain-specific instructions – specifically, cooking recipes – to convert them into flow graphs. We then present an approach to learn embeddings for entities and actions that occur in the flow graphs such that we can use the embeddings to identify plausible modifications that can be made to the instructions. Intuitively, our approach aims to learn embeddings that capture the flow of entities and actions from the flow graph, which in turn can be used to dynamically compute the output of a recipe after performing an ingredient substitution.

Our contributions are as follows: **(1)** Present a rule-based method to generate flow graphs from instruction text, leveraging domain ontologies and dependency parsing tools. **(2)** Introduce a novel graph embedding strategy for flow graphs, which aggregates nodes to better capture instruction steps and dynamically calculates intermediate results. Our method requires learning embeddings for significantly fewer nodes compared to baseline graph embedding models while showing top performance at a link prediction task for cooking recipes. **(3)** Present a method to identify plausible entity substitutions in flow graphs using our embedding calculation approach. Further, this method can handle new combinations of entities and actions without additional training.

2 Problem Formulation

Here, we give a brief overview of our main problem formulation and definitions. While this work focuses specifically on recipes, the approach can be extended to procedural instructions in different domains in a similar manner.

2.1 Recipe Modeling

A recipe R contains two pieces of information – a list of steps in natural language, S_R, and the set of ingredients used in the recipe, I_R. $S_R = [S_i | i = 1..n]$ is a list of

individual sentences, ordered sequentially as in recipe steps. Each ingredient $I_j \in I_R$ is a distinct ingredient defined by the recipe. We represent the ingredients and recipes following Resource Description Framework (RDF) standards to enable better integration with ontology and knowledge graph resources.

2.2 Flow Graph Representation

A key property of procedural instructions is that the main task of the instructions is to create an output entity through some combination and transformation of input entities. A recipe takes raw ingredients, applies transformations (such as cutting) to them, and combines them to form the final dish. Transformation that are applied may change properties of the original inputs (such as "diced tomatoes"), and the instructions provide us with a trace of how such intermediate results were formed. As such, it is sensible to consider representing instructions as a "flow" that captures how input items are processed through the instructions.

Our goal is to parse the instructions with the set of ingredients contained in R to form a flow graph. For this work, we define a flow graph as follows:

Definition 1. *A* ***flow graph*** *is an RDF graph of triples* `(h,r,t)`*, denoting a relation* `r` *from entity* `h` *to entity* `t`*, with the following properties: (1) the graph contains no cycles; (2) the graph has a single output node that is reachable by all other nodes; (3) all incoming relations to a node have the same label; and (4) all domain-specific entities have no incoming relations.*

In our definition, we distinguish domain-specific entities as equipment or ingredients that are specified in the recipe text. All such entities act as leaf nodes in the flow graphs. Other nodes in the flow graph, which have incoming relations, are denoted as *intermediate nodes*. In turn, the relations in the flow graphs correspond to the actions taking place in the recipe instructions, and their connections and directions indicate how the entities and intermediate nodes are being processed through the flow graph.

Example 1. Consider our running example in Fig. 1. This flow graph contains no cycles and has a single output node G. All incoming relations to intermediate nodes also share the same label. Lastly, all entities corresponding to ingredients or cooking equipment are leaf nodes. We also can observe how the edge labels correspond to actions taking place in the recipe.

Our use of the terminology "flow graph" resembles that of some prior works [7,15,27], but we make several distinctions surrounding what information is captured and how it is represented. Our requirement that entities must be leaves in flow graphs is not shared by previous definitions. Additionally, prior models do not have restrictions that incoming edges must have the same label, and intermediate results (as we model in our flow graph definition) are not modeled.

We also note the omission of several details from our example recipe's instructions. For example, the details to cook the bacon "over medium high heat" and "until crispy" are omitted in our flow graph. For the scope of this work, we

chose to focus on capturing and using the core information about actions and entities while dropping additional qualitative modifiers. Another point of omission is information about what role each entity plays in an action, as in how the bacon is being placed *into* the skillet. For the scope of this work, we simplify this information to only capture which entities were involved in the action. In these omissions, we opted to favor simplification of the flow graph at the cost of semantic accuracy due to the difficulty of correctly parsing the instructions.

Lastly, we omit information from sentences that are unrelated to cooking the actual recipe. Whenever ingredients that weren't included in the recipe's ingredient list occurred, we considered it extraneous information.

3 Flow Graph Generation from Instructions

To construct flow graphs, we make use of natural language processing (NLP) tools, a part-of-speech (PoS) tagger and dependency parser, as well as ontologies to provide knowledge about domain-specific entities. After using such tools to extract relations between entities and actions from each step in the recipe, the steps are combined together to form a flow graph. We note that our data and methods focus only on handling English recipe texts.

3.1 Parsing Instruction Text

The first step we apply is to perform dependency parsing and PoS tagging over each sentence in the recipe's instructions. Our goal is to find verbs and their associated nouns; these verbs are the actions taking place in the instructions. In our experiments, we perform this step using spaCy's [9] pretrained language models. An example of the dependency tree that is produced by spaCy can be seen in Fig. 2. Based on both the PoS and dependency tags produced by spaCy, we devised a rule-based method[1] to connect nouns and verbs occurring in each sentence to serve as the foundation for forming the recipe's flow graph.

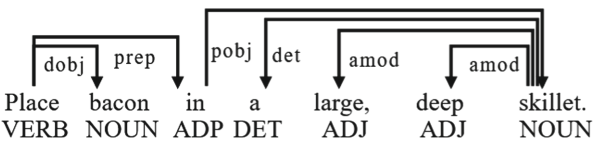

Fig. 2. An example dependency tree produced for a sentence using spaCy.

After processing each step in the recipe, we are left with a list of information pertaining to verbs and nouns that are directly interacting with each other in each step of the recipe. For example, from the example sentence in Fig. 2, we extract two tuples of verb-noun relations – ("place", "bacon"), and ("place",

[1] We refer to Sect. 1 in our supplemental material for further details on this process.

"large deep skillet"). The dependency relation between each verb and noun is retained for use in forming the flow graph. We discarded prepositions as well as adverbs to omit some details as noted in the previous section.

Correcting Parses: We found that parsing errors would occur frequently for sentences that were particularly terse or used implicit subjects (e.g., "Brown beef in the pot."). Such sentences have the subject (i.e., the person cooking the recipe) omitted, as is typical with many imperative sentences, and were ambiguous in how they should be parsed (e.g., "brown" may be a verb or an adjective).

We expect each sentence in the instructions to provide some meaningful action to perform, so in cases where no verb is found, we re-run the dependency parse with an augmented version of the sentence. Instructions are often presented as imperative sentences, and in English the imperative mood is typically (1) in the present tense and (2) in the second person. Based on this knowledge, in practice we found that simply adding a subject – the word "you" – to the beginning of the sentence resolved many such errors. For example, "you brown beef in the pot." resulted in correctly tagging "brown" as a verb.

Filtering. To better focus on modeling objects that are relevant to the instructions, we can use a domain-specific ontology to filter out extraneous information. In our experiments we use FoodOn [3], an ontology containing information about thousands of different foods and their relations, and filter out entities. We do this by matching noun-phrases from recipe texts to FoodOn classes based on their class labels, alternative names, and synonyms. We convert all noun-phrases and class names into one-hot vectors, weighted by TF-IDF measures, and calculate their cosine similarity to determine matches. In cases where a sufficiently high-confidence match was not found, we consider the noun irrelevant for our task and discard the information. We also retain links between ingredients in flow graphs and FoodOn classes for later use to train embeddings.

3.2 Forming Flow Graphs

After parsing instructions, we have a list of tuples containing verbs, nouns, and their relations in each step. We proceed to form a flow graph of the overall recipe by forming small graphs for the content of each step and then connecting the graphs for each step together into a single flow graph.

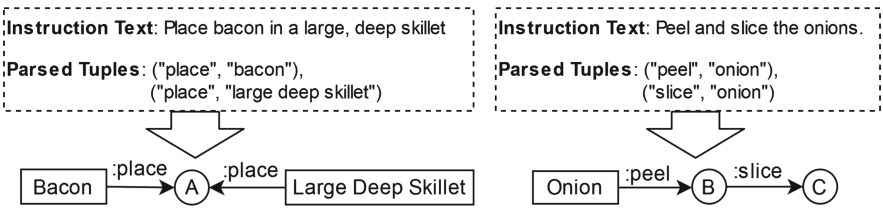

Fig. 3. Examples illustrating how two entities can form a single output (left) and how multiple verbs are applied sequentially to an entity (right).

First, to form minimal graphs from each step, we use verb-noun relations that were detected from the dependency parser. The verb is used as the edge label to connect the nouns to an output node. In cases where multiple verbs were used in the step, we assume that the noun and intermediate node content in the step are connected sequentially (as they occur in the step's sentence). An example of this step can be seen in Fig. 3.

Using the minimal graphs from each step in the instructions, we move on to connect each step together to form the overall flow graph. We consider 3 cases: (1) a step includes a reference to an entity that has been used in a previous step; (2) a step's dependency parse includes a verb with no direct subject or object; and (3) a step follows sequentially from the previous step.

Case 1: In the first case, we check for noun occurrences in each step to see if the same ingredient is being used. If such a situation exists, we connect the two steps together by adding an edge from the output of the earlier step to the first intermediate node in the later step. We check each step in order, prioritizing earlier steps when adding connections. An example demonstrating how two steps would be connected in this kind of case can be seen in Fig. 4.

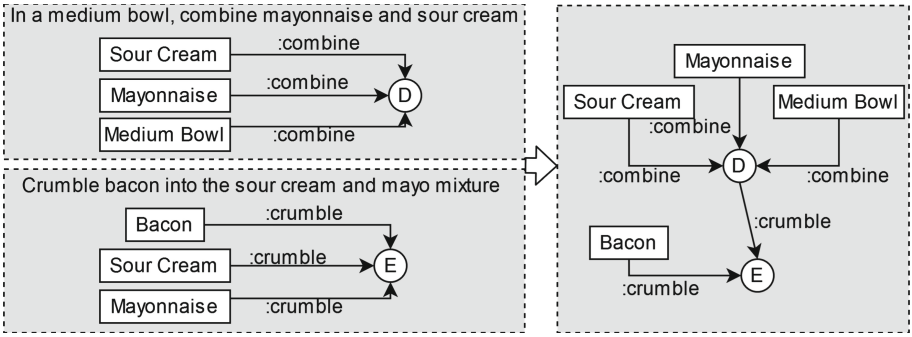

Fig. 4. An illustration connecting two steps in the running example together.

Case 2: For the second case, we use dependency relations between words that were obtained from the dependency parser. If the step includes a verb but has no relations to a direct subject or direct object, we infer that the verb is acting on the output of the previous step. An example of this situation can be seen in the first two steps of our running example, as "(1) Place bacon in a large, deep skillet", "(2) Cook over medium high heat until crispy". We can infer that the second step means we must cook the bacon – in the dependency parse result, "cook' in step 2 contains no direct subject or object.

Case 3: If either of the previous two cases do not apply, we simply connect steps together sequentially. In this case, the output node of each step is connected to the first intermediate node in the next step. The edge for this connection copies the same label as other incoming edges for that intermediate node, since we

can assume that the output of the step is having the same actions applied as other entities in that step. Sequential connections have been shown to be a good baseline for creating flow graphs in the domain of cooking [10].

Recipe-Specific Cases: Another consideration for a recipe's flow graph is that we expect to see all of the ingredients specified by the recipe. While this sometimes is trivial, there are often cases where ingredients are referred to by alternate names within the recipe steps or as a group of ingredients (e.g., instructions to "add herbs" rather than individually listing out each herb). In cases where not all relevant ingredients from the recipe have been included, we identify leaf nodes in the flow graph that are most similar to the missing ingredients. We used a measure of semantic similarity, wpath [31], over FoodOn's ingredient class hierarchy to determine which node is the most similar.

We also must consider a recipe-specific special case for phrases such as "all ingredients" and "remaining ingredients." These phrases occur fairly often in recipes written by non-experts and rely on the assumption that we know all ingredients in the recipe ahead of time. They also rely on sequential knowledge about which ingredients have already been used in the recipe. When either of these cases occur, we check the flow graph for all instances of ingredient usage in prior steps and add new edges for any ingredients that have not been used yet as the "remaining" ingredients.

4 Flow Graph Embedding

A key motivation for using flow graphs in our work is to enable us to view actions that take place in the instructions as transformations on the input nodes. This perspective is similar that of common translational knowledge graph embedding (KGE) techniques, such as TransE [2]. Given a triplet `(h,r,t)`, such KGE methods model `t` to be the result of applying some transformation `r` on `h`. In TransE, embeddings are learned such that $\mathtt{h} + \mathtt{r} \approx \mathtt{t}$ given $\mathtt{h}, \mathtt{r}, \mathtt{t} \in \mathbb{R}^k$. In this way, the relation `r` is used as a transformation on the entity `h` to produce the result entity `t`. Extending this idea to our flow graphs, our aim is to model our relations – i.e., actions such as "cook" or "crumble" – as transformations on the input ingredients to produce output intermediate nodes.

However, our flow graphs for procedural instructions are not well suited to directly apply KGEs that are trained over triplets of data. While KGE models view each triple independently as indicating a single factual statement, in our flow graphs *all* of the incoming nodes contribute to the output. Additionally, standard KGE model training over triples would require us to learn embeddings for all intermediate nodes, which is undesirable for our case as the number of unique intermediate nodes rapidly increases with the number of flow graphs.

4.1 Embedding Strategy

To address the aforementioned issues, we incorporate the idea of performing aggregation on incoming nodes in the flow graph. This aggregation should serve

to provide additional context when training embeddings such that all ingredients involved in a recipe's step are considered while training. Additionally, we address the issue of handling intermediate nodes by calculating the output of applying transformations (based on relation embeddings) to entity embeddings during each training step. Figure 5 illustrates how entity embeddings are aggregated and calculated (calculations past node C are omitted for brevity). The aggregation is performed by taking the mean of the input nodes, while the output is calculated similar to TransE's $h + r = t$ formulation. Leveraging the fact that all incoming edge labels in our flow graphs are the same for a given intermediate node, each aggregation is treated as a single "head" entity in the KGE model's $(\mathtt{h}, \mathtt{r}, \mathtt{t})$ triplet, and the embeddings for intermediate nodes are calculated on the fly.

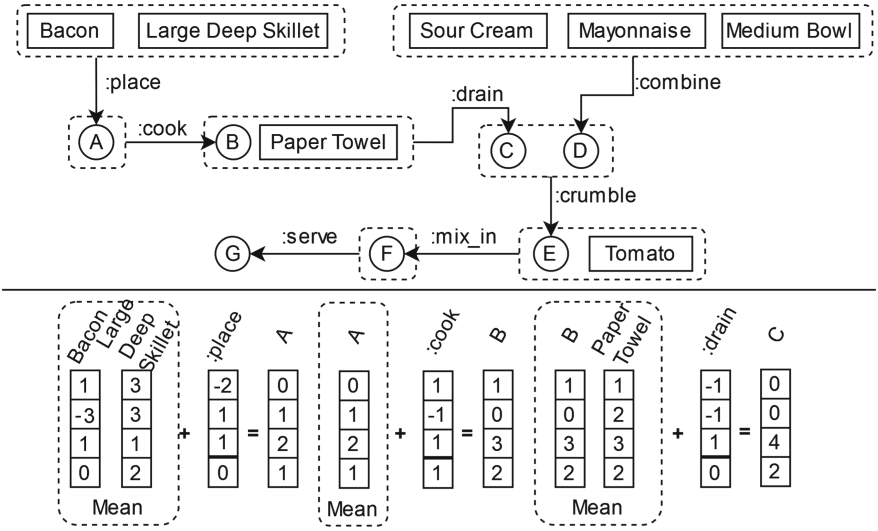

Fig. 5. An illustration of how we aggregate input nodes, within the dotted lines, and apply the embedding of the action to produce intermediate nodes.

Distance: We define the distance metric used during each training step in our model in a recursive fashion by defining a "triplet" (h_R, r_R, t_R), where $h_R, r_R, t_R \in \mathbb{R}^k$, for each recipe R. Our goal during training is to minimize the distance $|h_R + r_R - t_R|$, following from the distance formulation of TransE which minimizes the distance $|h + r - t|$.

Given a flow graph F_R, let I_v denote the set of nodes with incoming edges to node v and l_v denote the label for incoming edges to node v. For the flow graph F_R and its output node v_o, we can then define h_R as the output of Algorithm 1. For the given recipe, $r_R = r_{l_{v_o}}$ is then defined as the last action that takes place in the recipe, and $t_R = h_{v_o}$ is the embedding of the recipe's output node.

Example 2. Applying Algorithm 1 to our running example recipe, the output node v_o=G. The incoming nodes $I_G = [F]$, so we can calculate the recipe's

embedding as $h_R = Aggregate([RecursiveAgg(F)])$. Stepping through the procedure for *RecursiveAgg*, we will reach line 5 where *Recursive Agg* is called again on the incoming nodes to F, $I_F = [E, Tomato]$. Tomato is a leaf node, while E will once again enter a recursive call which we omit for brevity. Back to node F, in line 6 we will use node F's incoming edge ":mix_in" and its embedding $r_{:mix\_in}$, and return $h_R = r_{:mix\_in} + Aggregate(mean([h_{Tomato}, RecursiveAgg(E)]))$. This value is then used with the output node's incoming edge, $r_{:serve}$, to calculate $h_R + r_{:serve}$ as this recipe's calculated output embedding value.

Training Objectives: The distance between the recipe's calculated "triplet" (h_R, r_R, t_R) is then computed as $dist_R = |h_R + r_R - t_R|$. Following standard training for KGE models using this distance metric, the loss is optimized as $L_p = -\log \sigma(\gamma - dist_R)$, where γ is a fixed margin and σ is the sigmoid function.

Algorithm 1. Flow Graph Output Embedding Calculation Pseudocode

Input A flow graph's output node v_o, incoming nodes I, incoming edge labels l
Output Calculated head vector $h_R \in \mathbb{R}^k$

```
1: function RecursiveAgg(v)
2:     if v.isLeafNode then
3:         return h_v
4:     else
5:         inNodes = [RecursiveAgg(v_j) for v_j ∈ I_v]
6:         return Aggregate(inNodes) + r_{l_v}
7:     end if
8: end function
9: function Aggregate(EmbeddingList)
10:    return mean(EmbeddingList)
11: end function
12: h_R = Aggregate([RecursiveAgg(v_j) for v_j ∈ I_{v_o}])
```

We additionally follow best practices for training KGE models by utilizing negative sampling. For a given recipe R, negative sampling is performed for an incorrect tail entity $t_{R'} \neq t_R$ and an incorrect "head" flow graph $h_{R'} \neq h_R$. $t_{R'}$ entity points to another randomly selected recipe output, and $h_{R'}$ is constructed by randomly replacing input nodes in R's flow graph. k negative samples were collected for each training step, and the negative sampling loss was calculated for the negative head and tail samples as $L_n = -\frac{1}{k}\sum_1^k \log \sigma(|h_R + r_R - t_{R'}| - \gamma) - \frac{1}{k}\sum_1^k \log \sigma(|h_{R'} + r_R - t_R| - \gamma)$. The total loss is calculated as $L = L_p + L_n$.

By using our recursive aggregation strategy, we can calculate embeddings for intermediate nodes rather than learning them explicitly. The only nodes in our flow graph data that we learn embeddings for are the ingredient leaf nodes and the recipe's final output node v_o.

In order to incorporate external domain-specific knowledge, we also include triples from FoodOn to perform training. We connect classes from FoodOn to

ingredients in our recipe dataset, identified during the flow graph generation stage, and perform normal training of the TransE model over this data.

4.2 Replacement Techniques

Once our entity embeddings $h \in \mathbb{R}^k$ and relation embeddings $r \in \mathbb{R}^k$ have been trained, we can apply the same aggregation techniques used during training – to calculate the "output" embedding, transforming the inputs – to perform modification and substitution of entities in a recipe.

Given a recipe's flow graph F_R, our model will have learned an embedding for the recipe's final output, h_{v_o}. Additionally, we can use the entity and relation embeddings for the nodes and edges in F_R to calculate the recipe's output as well (once again following from the intuition that the embeddings $h_R + r_R = t_R$). The original recipe's learned output node embedding, h_{v_o}, and the calculated output embedding of the original recipe's flow graph, t_R, can be used to identify plausible substitutions of ingredients by replacing nodes in F_R and calculating a new output embedding.

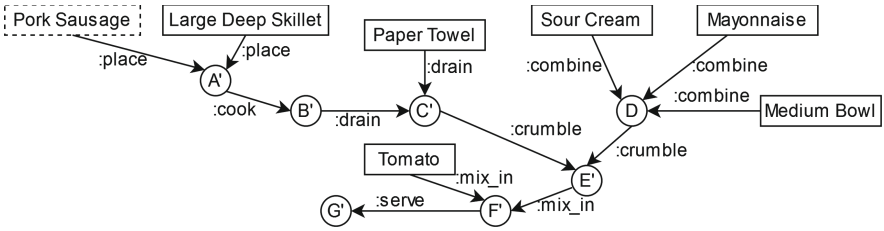

Fig. 6. Substituting "Bacon" with "Pork Sausage" in our running example recipe.

For an ingredient node $v \in F_R$ that we wish to replace, we simply can swap v with a new node v_s as seen in Fig. 6. We also replace all edges in F_R to which v was connected. Then, following the procedure from Algorithm 1, we can calculate a new output embedding for the flow graph with a node substitution as $t_{R'}$. To determine whether the substitution seems "good" or not, we can then compare the cosine similarity of the newly calculated embedding $t_{R'}$ with the original learned embedding h_{v_o} or the original calculated embedding t_R. This process can then be repeated over a number of substitute ingredient options to produce a ranking of which substitute is the "best" based on how similar the newly calculated result is to the original.

A result of our embedding and substitution strategy is that it is robust in its ability to handle previously unseen recipes. Assuming that embeddings have been learned for the relevant ingredients and actions, the output embedding for a new recipe's flow graph is dynamically calculated and would require no additional training. A completely novel recipe can therefore have an embedding representing its output, which in turn allows us to perform our substitution strategy.

5 Evaluation

5.1 Flow Graph Generation

We evaluate the quality of our flow graph generation method by comparing against a dataset of recipe annotations and flow graphs published by Yamakata et al. [27]. However, the level of detail, included concepts, and formulation of their flow graphs differs from that of our work. We therefore performed preprocessing, which included adjusting the graph's connections so that actions were edges rather than nodes.[2] After performing our preprocessing step, we evaluate F-measure by comparing edges in our generated graph versus the ground truth.

5.2 Embedding Flow Graphs

To evaluate the quality of our embeddings, we frame our problem as a knowledge graph completion task for individual flow graphs. Given a recipe flow graph and the learned embedding for its output node, we remove a single ingredient and then rank the ingredient that is most likely to fit in to the flow graph. Following our ingredient substitution procedure from Sect. 4.2, candidate ingredients are used as "substitutions" to calculate recipe output embeddings, and their similarity to the expected embedding of the recipe is used to rank ingredients. Our goal for this experiment is to demonstrate that our EaT-PIM method can effectively re-identify a missing ingredient from a recipe, which in turn would suggest that we might be able to identify plausible substitutions by selecting ingredients that are similar to the "missing" ingredient being replaced.

Dataset: We conduct our experiments using recipe data from Food.com [14]. We randomly selected a subset of the data consisting of 20,000 recipes, which included 6,142 distinct ingredients. We generated flow graphs for each recipe, and this data was further split into training, validation, and test data using a 70%, 15%, 15% split. Embeddings were trained using data from the training set as well as data from FoodOn [3], which similarly was split for training.

Baselines: Our first baseline uses a simplified problem setup, which omits the flow graph data and instead ranked missing ingredients based on ingredient co-occurrence in recipes (denoted COOC). The sum of co-occurrence probabilities between a candidate ingredient of all ingredients in a target recipe was used to produce a score, which was then used to rank the missing ingredient.

Our next set of baselines utilize standard KGE models. To enable training over triplets of data, we re-introduce explicit intermediate nodes for flow graphs in these baselines. We train two translational distance models, TransE [2] and RotatE [22], which are well suited for modeling compositional relations. We contrast these with two semantic matching models, DistMult [29] and ComplEx [23], which are better suited for modeling symmetric and antisymmetric

[2] We refer to Sect. 2 in our supplemental material for details on the preprocessing.

relations. RotatE and ComplEx also learn embeddings in complex vector space. These four baselines were trained using embedding sizes of 200, 300, and 400 dimensions, and the best results are reported. Lastly, we train a graph neural network (GNN)-based embedding model from [17] (denoted GNN (Nathani et al.)), which uses a Graph Attention Network [24] together with a convolutional layer [18] to perform link prediction. For each model, we perform link prediction for intermediate nodes connected to input ingredient and rank the missing ingredient to calculate performance. This ranking is performed in a filtered setting – i.e., the ranking is not penalized if a *true* triple is highly ranked.

For our final baseline, we introduce an additional TransE-based baseline that is trained in a similar manner to EaT-PIM (denoted TransE (flow graph path)). Rather than explicitly learning embeddings for intermediate nodes, this model is trained by using the path of edges between ingredients and the recipe output. This model differs from EaT-PIM in that no node aggregation is performed.

5.3 Results

Flow Graph Generation: EaT-PIM's methods to convert recipe texts to flow graphs yielded a precision of **0.638**, recall of **0.566**, and F1 score of **0.600** when comparing them to the ground-truth graphs. To give a rough comparison (albeit for a slightly different task[3]), the original results reported by Yamakata et al. [28] indicate an F1 of 0.433 for their full pipeline. Considering that our methods did not require any annotated training data, our results appear competitive with those presented in the original dataset publication.

Embedding Flow Graphs: Table 1 displays the mean reciprocal rank (MRR), HITS@3, HITS@5, and HITS@10 for our EaT-PIM method and the baselines. MRR is calculated as the average of $1/rank_t$, where $rank_t$ is the rank of the true entity t for each datapoint. HITS@K is calculated as the proportion of inputs for which the correct entity t is within the top K ranks.

Table 1. Results for ranking missing ingredients in recipe flow graphs.

Model	MRR	HITS@3	HITS@5	HITS@10
COOC	0.132	0.138	0.189	0.281
DistMult	0.012	0.012	0.015	0.021
ComplEx	0.017	0.018	0.023	0.032
RotatE	0.118	0.120	0.163	0.242
TransE	0.151	0.158	0.211	0.301
GNN (Nathani et al.)	0.068	0.068	0.88	0.124
TransE (flow graph path)	0.172	0.177	0.206	0.254
EaT-PIM (ours)	**0.286**	**0.355**	**0.437**	**0.520**

[3] Further details are discussed in Sect. 2 of our supplemental material.

EaT-PIM is able to outperform the baselines by a large margin for the task of re-identifying missing ingredients from recipes. Surprisingly, we find that the basic TransE model shows the best performance among our standard KGE baselines, followed by RotatE. The ability for these two models to capture compositional relations shows a stark contrast in performance compared to DistMult and ComplEx, which appear to be poorly suited for our task.

The TransE (flow graph path) baseline shows the second best performance. Compared to the standard TransE model, performing training and predictions based on the entire path from the ingredient to recipe output appears to have provided minor benefits. Our approach to perform aggregation improves upon this further – when applying the embedding trained through EaT-PIM to perform ingredient prediction only based on the path from the ingredient to the recipe, the MRR increases to 0.260. This suggests that EaT-PIM's approach to aggregate nodes was particulary useful to learn good embeddings, while applying EaT-PIM's substitution method to perform the link prediction granted an additional 10% increase in performance.

Discussion: EaT-PIM's ability to dynamically compute intermediate nodes allows it to learn embeddings for significantly fewer entities than standard KGEs require (40,500 entities in EaT-PIM versus 272,000 in baselines). This can be beneficial during training, as less memory is needed to load all of the embeddings. Additionally, EaT-PIM's simple model is less resource intensive compared to more advanced models such as the GNN. The GNN in our experiment required 190 MB of memory to store 50 dimensional embeddings of nodes along with the convolutional neural network, while EaT-PIM's 200 dimensional embeddings only needed 30 MB. This benefit would increase further if training is performed for more recipes, suggesting strong potential for scalability using EaT-PIM.

Table 2. Examples of top ranked substitutions in two recipes.

Recipe	Target ingredient	Top 3 substitutes
Pork marinate	Pork	Boneless pork, Rib, Pork loin roast
Mashed potatoes	Red potato	Dried thyme, all purpose flour, Chicken

Regarding our application of these embeddings to ingredient substitutions, while it is challenging to evaluate due to subjectivity issues, we observe that using EaT-PIM to rank substitutions generally produces reasonable results. Table 2 shows examples of ranking substitutions for a target ingredient in a specific recipe. The top substitutions for "Pork" are all varieties or names of pork. On the other hand, we observe some less desirable substitutes, such as thyme, for "Red Potato" in our example. While work remains to improve the consistency of substitution ranking, our methods can provide some utility by comparing substitutions across different recipes. For example, Table 3 displays the relative

rankings of three potato substitutes[4] in different types of recipes.[5] While it is difficult to judge how *correct* these relative rankings are, it demonstrates that the suitability of each ingredient varies based on the recipe at hand.

Table 3. A comparison of relative rankings of potato substitutes in three recipes.

Recipe: **Mashed Potato**	Recipe: **Potato Gratin**	Recipe: **Healthy Soup**
Substitute Ranks	Substitute Ranks	Substitute Ranks
1. Jicama	1. Cauliflower	1. Rutabaga
2. Cauliflower	2. Jicama	2. Cauliflower
3. Rutabaga	3. Rutabaga	3. Jicama

6 Related Work

Ingredient Substitution: Previous works on ingredient substitution have explored methods such as rule-based substitutions in TAAABLE [5] and Intellimeal [21]. DIISH [20] applied a substitutability heuristic based on ingredient co-occurrence and similarity. A major limitation of such works was that they did not explicitly incorporate detailed information about cooking instructions.

Workflow Extraction: Extracting workflows instructions has been explored in the domain of cooking using methods such as frame- and pattern-based extraction [19] and case-based reasoning [4]. Semantic representations of procedural knowledge were proposed in [30], including annotations of pre-conditions, actions, and purpose. Outside of the cooking, explicit representations of procedural instructions have been investigated a variety of domains [1,6,12,16]. Our work shares some similarities to prior works in the use of ontologies to identify relevant entities. However, we do not rely on manually constructing templates to extract workflows, and the flow graph representation of our methods also differs.

Flow Graphs: The flow graphs modeled in our work shares similarities with past works such as [13,15,27,28]. Many previous works using recipe flow graphs use annotations [13,25,27], either by directly using the annotations or learning to predict labels and relations based on a training set, while our work does not rely on annotated data. A method demonstrated in [10] formed flow graphs in an unsupervised fashion, but it relied on an external parser to classify words.

Knowledge Graph Embedding: Beyond the baseline models applied in our experiments [2,22,23,29], a variety of distance metrics have been proposed for training KGE models [26]. Such models treat triples in the graph as independent facts, while our motivation of applying them to flow graphs would want to consider the combination of *multiple* triples together to produce an output.

[4] Jicama and rutabaga are often cited as healthy potato substitutes.

[5] We refer to Sect. 3 in our supplemental material for details on the example recipes.

Embedding graphs using graph neural networks (GNN), such as [8,11,24], have also gained traction in recent years. GNNs have demonstrated the benefits of aggregating information from neighboring nodes. Our embedding approach takes inspiration from such methods in that we also aim to aggregate input nodes.

7 Conclusion

We present EaT-PIM, which consists of two main methods. First, EaT-PIM converts procedural instructions into flow graphs using NLP tools and domain-specific ontologies. Using the generated flow graphs, EaT-PIM trains an embedding model using a strategy that allows us to aggregate input information and dynamically compute intermediate node representations within flow graphs. Our evaluations demonstrate strong performance of EaT-PIM in both generating flow graphs and performing link prediction for ingredients in recipes. Future work includes exploration of more intricate aggregation strategies in the embedding and applying EaT-PIM to instructions from different domains to explore substitutability for more diverse types of entities.

Supplemental Material Statement: Supplemental materials and source codes are made available at https://github.com/boschresearch/EaT-PIM.

Acknowledgements. We would like to express our thanks to the colleagues of Bosch's RTC-NA, the members of RPI's Tetherless World Constellation, and CMU's Naoki Otani for their feedback and reviews of this manuscript.

References

1. Agarwal, S., Atreja, S., Agarwal, V.: Extracting procedural knowledge from technical documents. ArXiv abs/2010.10156 (2020)
2. Bordes, A., Usunier, N., García-Durán, A., Weston, J., Yakhnenko, O.: Translating embeddings for modeling multi-relational data. In: NIPS (2013)
3. Dooley, D.M., et al.: Foodon: a harmonized food ontology to increase global food traceability, quality control and data integration. NPJ Science of Food 2 (2018)
4. Dufour-Lussier, V., Ber, F.L., Lieber, J., Meilender, T., Nauer, E.: Semi-automatic annotation process for procedural texts: an application on cooking recipes. ArXiv abs/1209.5663 (2012)
5. Gaillard, E., Lieber, J., Nauer, E.: Adaptation of taaable to the ccc'2017 mixology and salad challenges, adaptation of the cocktail names. In: ICCBR (Workshops), pp. 253–268 (2017)
6. Halioui, A., Valtchev, P., Diallo, A.B.: Ontology-based workflow extraction from texts using word sense disambiguation. bioRxiv (2016)
7. Hamada, R., Ide, I., Sakai, S., Tanaka, H.: Structural analysis of cooking preparation steps in Japanese. In: IRAL 2000 (2000)
8. Hamilton, W.L., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: NIPS (2017)

9. Honnibal, M., Montani, I., Van Landeghem, S., Boyd, A.: spacy: Industrial-strength natural language processing in python (2020)
10. Kiddon, C., Ponnuraj, G.T., Zettlemoyer, L., Choi, Y.: Mise en place: unsupervised interpretation of instructional recipes. In: EMNLP (2015)
11. Kipf, T., Welling, M.: Semi-supervised classification with graph convolutional networks. ICLR (2017)
12. Kulkarni, C., Xu, W., Ritter, A., Machiraju, R.: An annotated corpus for machine reading of instructions in wet lab protocols. In: NAACL (2018)
13. Maeta, H., Sasada, T., Mori, S.: A framework for procedural text understanding. In: IWPT (2015)
14. Majumder, B.P., Li, S., Ni, J., McAuley, J.: Generating personalized recipes from historical user preferences. In: EMNLP-IJCNLP, pp. 5976–5982. Association for Computational Linguistics, Hong Kong, China, November 2019
15. Mori, S., Maeta, H., Yamakata, Y., Sasada, T.: Flow graph corpus from recipe texts. In: LREC (2014)
16. Mysore, S., et al.: Automatically extracting action graphs from materials science synthesis procedures. CoRR abs/1711.06872 (2017). http://arxiv.org/abs/1711.06872
17. Nathani, D., Chauhan, J., Sharma, C., Kaul, M.: Learning attention-based embeddings for relation prediction in knowledge graphs. In: ACL (2019)
18. Nguyen, D.Q., Nguyen, T.D., Nguyen, D.Q., Phung, D.Q.: A novel embedding model for knowledge base completion based on convolutional neural network. In: NAACL (2018)
19. Schumacher, P., Minor, M., Walter, K., Bergmann, R.: Extraction of procedural knowledge from the web: a comparison of two workflow extraction approaches. WWW (2012)
20. Shirai, S.S., Seneviratne, O., Gordon, M.E., Chen, C.H., McGuinness, D.L.: Identifying ingredient substitutions using a knowledge graph of food. Front. Artif. Intell. **3**, 111 (2021)
21. Skjold, K., Øynes, M., Bach, K., Aamodt, A.: Intellimeal-enhancing creativity by reusing domain knowledge in the adaptation process. In: ICCBR (Workshops), pp. 277–284 (2017)
22. Sun, Z., Deng, Z., Nie, J.Y., Tang, J.: Rotate: Knowledge graph embedding by relational rotation in complex space. ICLR (2019)
23. Trouillon, T., Welbl, J., Riedel, S., Gaussier, É., Bouchard, G.: Complex embeddings for simple link prediction. In: ICML (2016)
24. Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio', P., Bengio, Y.: Graph attention networks. ICLR (2018)
25. Wang, L., Li, Q., Li, N., Dong, G., Yang, Y.: Substructure similarity measurement in Chinese recipes. In: WWW (2008)
26. Wang, Q., Mao, Z., Wang, B., Guo, L.: Knowledge graph embedding: a survey of approaches and applications. IEEE Trans. Knowl. Data Eng. **29**, 2724–2743 (2017)
27. Yamakata, Y., Imahori, S., Maeta, H., Mori, S.: A method for extracting major workflow composed of ingredients, tools, and actions from cooking procedural text. In: 2016 IEEE International Conference on Multimedia Expo Workshops (ICMEW), pp. 1–6 (2016)
28. Yamakata, Y., Mori, S., Carroll, J.: English recipe flow graph corpus. In: LREC (2020)
29. Yang, B., tau Yih, W., He, X., Gao, J., Deng, L.: Embedding entities and relations for learning and inference in knowledge bases. CoRR abs/1412.6575 (2015)

30. Zhang, Z., Webster, P., Uren, V.S., Varga, A., Ciravegna, F.: Automatically extracting procedural knowledge from instructional texts using natural language processing. In: LREC (2012)
31. Zhu, G., Iglesias, C.A.: Computing semantic similarity of concepts in knowledge graphs. IEEE TKDE **29**(1), 72–85 (2017)