Flow graph to Video Grounding for Weakly-supervised Multi-Step Localization Supplemental Material

Nikita Dvornik, Isma Hadji, Hai Pham, Dhaivat Bhatt, Brais Martinez, Afsaneh Fazly, and Allan D. Jepson

Samsung AI Center

1 Summary

Our supplemental material is organized as follows: Section 2 elaborates on our approach to construct tSort graphs from flow graphs and presents an efficient implementation to tackle this step of our Graph2Vid approach. We then present a detailed complexity analysis of Graph2Vid compared to the brute-force approach in Section 3. In Section 4, we present details on both the rule-based and learning-based parsers that we used to convert procedural text to flow graphs. We elaborate on our experimental setup in Section 5. Finally, in Section 6, we provide additional analysis of flow graphs and their influence on step localization performance.

2 Efficient algorithm for tSort graph construction

In Section 3.3 of the main paper we presented a simple algorithm for tSort graph construction. Here, we further elaborate on tSort graph construction and present the more efficient procedure implementation (actually used in our work). In Algorithm 1, we present the algorithm used in our implementation to construct the tSort graphs, $\mathcal{S}$, given the flow graph, $\mathcal{G}$. The algorithm uses Breadth First Search (BFS) traversals in $\mathcal{G}$, starting from the sink of the graph and following the edges of $\mathcal{G}$ in the opposite direction (moving backwards to the root). During the traversal, we build the tSort graph, $\mathcal{S}$, such that each node of $\mathcal{S}$ is a tuple (v, F), where $v \in V_G$ is a node in $\mathcal{G}$, and $F \subset V_G$ is the subset of nodes visited so far. Specifically, during the BFS traversal, v is the node that is currently being considered and it is referred to as the **"active node"**. On the other hand, F is the set of nodes that have been visited by the BFS traversal on separate threads (i.e., distinct from the thread containing the active node v) and they are collectively referred to as the **"front"**.

In the main paper, we explain that a tSort graph, $\mathcal{S}$, efficiently captures all the topological sorts of the graph, $\mathcal{G}$, while avoiding redundant paths. To achieve this property we check for path feasibility as described in Algorithm 1. In particular, to make sure that during the BFS traversal we explore only the paths that conform to the original flow-graph, $\mathcal{G}$, (i.e., valid topological sorts

Algorithm 1 tSort-graph Construction

```
 1: Inputs: $\mathcal{G}$- flow graph, $s$ - sink node in $\mathcal{G}$
 2: $E_{tSort}$ = []                                        ▷ init edge set of the tSort-graph
 3: $q$ = queue((s, set()))                                  ▷ init BFS queue
 4: while q do
 5:     $v, F = q$.pop()                                     ▷ active node v, set of visited front F
 6:     $P_v$ = get_predecessors($v$, $\mathcal{G}$)          ▷ Get predecessors of v in G
 7:     for $v_{new}$ in $P_v \cup F$ do
 8:         $F_{new} = P_v \cup F/\{v_{new}\}$
 9:
10:         is_feasible = True
11:         for $v_F$ in $F_{new}$ do                         ▷ Checking for feasibility of this path
12:             if lowest_common_ancestor($\mathcal{G}$, $v_{new}$, $v_F$) = $v_{new}$ then
13:                 is_feasible = False
14:
15:         if is_feasible = True then
16:             $q$.append(($v_{new}$, $F_{new}$))               ▷ Add new node to the queue
17:             $E_{tSort}$.add((($v, F$), ($v_{new}, F_{new}$)))  ▷ Add new edge to $\mathcal{S}$
18: $\mathcal{S}$ = build_graph_from_edges($E_{tSort}$)          ▷ build the tSort graph
19: Output: $\mathcal{S}$
```

of $\mathcal{G}$), we check that the active node v_{new} is *not* an ancestor of any node in the front F_{new} in the original graph, $\mathcal{G}$. To understand why, remember that the set F contains nodes already visited by the traversal, while v_{new} is the node being visited currently. Therefore, if v_{new} was an ancestor of one of the nodes $v_F \in F$ in $\mathcal{G}$, then v_F must be visited *after* v_{new} in any valid topological sort of $\mathcal{G}$. Thus, the transitions tuples (v_{new}, F_{new}) that do not satisfy the feasibility criteria above are not added to the traversal path (see Alg. 1, lines 10-13). In summary, using BFS for graph traversal (i.e., an algorithm that is guaranteed to list all possible traversals of a graph) combined with the feasibility criteria described above, guarantees that $\mathcal{S}$ contains *all* the valid topological sorts of $\mathcal{G}$. Fig. 1 illustrates our tSort graph construction as described in Algorithm 1.

3 On the algorithm's complexity

To develop some intuition on both the size of the generated tSort graphs, and on the speed-up over the naive approach described in Section 3.2 of the main paper, we consider simple model problems where the flow graph consists of T separate, linearly-ordered threads, with $n_1, n_2, \ldots, n_T \geq 1$ nodes in each thread, for a total of $\sum_{t=1}^{T} n_t = n$ steps. For simplicity, we also add a unique root node, s, to $\mathcal{G}$, with edges to the beginning of each of the T threads. We refer to such a flow graph as $\mathcal{G}(n_1, \ldots, n_T) = (V_G, E_G)$.

For two threads ($T = 2$), ignoring the root node for a moment, the topological sorts of the flow graph are called riffle shuffle permutations, specifically (n_1, n_2)-shuffles [8]. These are analogous to the permutations that can be obtained from

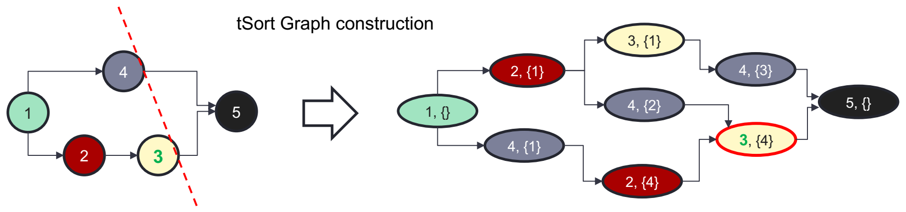


Fig. 1: tSort graph construction. (left) Original flow-graph $\mathcal{G}$. The red dashed line illustrates the visited "front" corresponding to the node of the tSort graph (3, {4}). (right) The obtained tSort graph $\mathcal{S}$. Every node in $\mathcal{S}$ is a tuple, where the first element is the active node in $\mathcal{G}$, and the second is the "front" of the traversal, i.e. the nodes last visited on all separate parallel threads. The node $(3, \{4\})$ ∈S corresponds to the node $3 \in G$ and the red dashed "front" intersecting the parallel thread at node 4.

a sorted deck of n cards by riffle shuffling a cut of the first n_1 with the remaining cards. There are $N_{tSort}(n_1, n_2) = \binom{n}{n_1}$ such riffle shuffles. This analysis is easily extended to show that the number of topological sorts (tSorts) for our model problems are given by

$$N_{tSort}(\mathcal{G}) = \frac{n!}{n_1! n_2! \dots n_T!}, \text{ where } n = \textstyle\sum_{t=1}^{T} n_t. \quad (1)$$

Note that N_{tSort} quickly becomes infeasibly large as n and T grow.

Next, for our model problems, we consider the number of nodes and edges in their tSort graph $\mathcal{S}(\mathcal{G}) = (V_S, E_S)$. From Algorithm 1 we see that any node in V_S is of the form (v, F) where v is the "active" node and F is the set of all nodes last visited by the traversal on the separate parallel threads, referred to as the "front". For our model problems, $|F| \leq T - 1$.

This form (v, F), clearly illustrates that, in this one node we are merging the prefix strings of all topological sorts that have arrived at node v having processed nodes in F `in any other order`. This merging of sequences to sets is the key to our efficiency gain.

Moreover, for our model problems, we can use the form (v, F) to count the number of nodes, $|V_S|$, in $\mathcal{S}$, along with the maximum in-degree of edges in E_S. Other than the root node, $(s, \emptyset)$, any node (v, F) is defined by picking a thread, t, and an active node, v, from the n_t nodes on that thread, and then forming F to characterize the state of processing in the other $T - 1$ threads. Specifically, for these other threads, we might not have started on that thread (in which case we need $s \in F$), or we have already processed down to a specific node in that thread. The total number of vertices is therefore seen to be

$$|V_S(\mathcal{G})| = 1 + \sum_{t=1}^{T} \left[n_t \prod_{j \neq t} (n_j + 1) \right], \text{ where } n = \textstyle\sum_{t=1}^{T} n_t,$$
$$= 1 + \left[\prod_{j=1}^{T} (n_j + 1) \right] \left[\sum_{t=1}^{T} \frac{n_t}{n_t + 1} \right]. \quad (2)$$

First, note that $|V_S|$ also rapidly grows with n and T. Therefore there will be practical limits to the size of flow graphs that are feasible to process in this manner, and in such cases we would need to resort to approximation approaches. However, in practice, we find that typical procedures result in tSort graphs of manageable sizes (see Section 4.5 of the main paper). Second, from Eq (2), we can note that the crude upper bound is $|V_S| = O(Tn^T)$. Hence, for a fixed number of threads T in our model problems, the number of nodes in the tSort graph is polynomial in the number of nodes, n, in the original graph. Finally, again for our model problems, the incoming edges at any node (v, F) in the tSort graph must be due to a single step being performed in the flow graph, which must have occurred in one of the T threads, either by advancing to the active state v from the previous state, or by advancing to an element in the front F in some other thread. That is, there must be at most T incoming (and, similarly, outgoing) edges to each node in $\mathcal{S}$.

The complexity of matching directly N_{tSort} topological sorts of a flow graph, $\mathcal{G}$, with $|V_G|$ nodes to a video of C clips, is $O(|V_G| N_{tSort}(\mathcal{G}) C)$, while the complexity of matching the associated tSort graph to C clips is $O(T|V_S(\mathcal{G})|C)$. The ratio of these leading order terms is therefore

$$\rho(\mathcal{G}) = \frac{N_{tSort}(\mathcal{G})|V_G|}{T|V_S(\mathcal{G})|}, \quad (3)$$

and this ratio gives a rough idea of the speed-up. In Fig. 2 we have plotted $log(\rho)$ as a function of n for various numbers of threads, $T \geq 2$, for the above model problems, where the number of elements in each thread is $n_t \in \{\text{floor}(n/T), \text{ceil}(n/T)\}$. We can clearly observe massive speed-ups for our problem setup.

4 Creating flow graph from procedural text

In this section we elaborate on the rule-based and learning-based graph parsers introduced in Section 3.6 of the main paper.

4.1 Rule-based graph parsing

Our flow graph construction pipeline is depicted in Figure 3. The pipeline consists of two main steps: text entity extraction and graph construction.

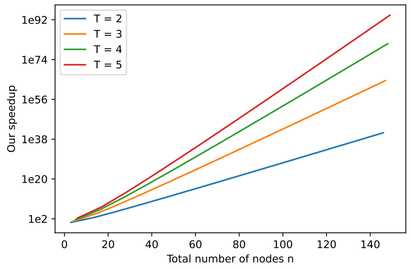


Fig. 2: Complexity speedup with Graph2Vid over the brute-force approach. The plot shows gains in complexity (given by Eq. (3)) in log scale for the model example in Sec 3 depending on the number of nodes n in the flow graph, and for different number of threads T.

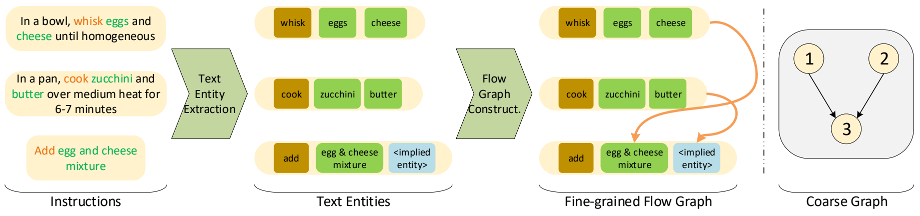


Fig. 3: The text-to-flow graph generation pipeline. It consists of two major steps: extracting text entities from the original instructions, and constructing the fine-grained graph, in which objects and actions are properly connected to form a complete procedural flow graph. Finally, this fine-grained graph is collapsed into a coarse sentence-level graph used in our graph-to-sequence grounding algorithm.

Text entity extraction. Starting from procedural text, we first extract relevant text entities from each individual sentence, including action verbs, direct and prepositional objects as suggested in [6]. Our entity extractor relies on an off-the-shelf dependency parser [7] in order to recover the verb and noun phrases from text. Furthermore, similar to previous work [3], we also take into account *"implicit objects"*, which are only implied from the text. For example, in the third sentence in Figure 3, the *"egg and cheese mixture"* is added to something that was omitted from the writing. One can deduce that this implicit object refers to the *"cooked zucchini"*, product of the action described in the second sentence. We implemented a set of specific rules on top of dependency parsing to augment the extracted entities with implicit entities. For instance, in the above example, we use the rule *"ADD [list of objects] TO [destination]"* to fill in the missing *destination* with the *"implicit object"*. These implied entities are also used in the graph construction.

Flow graph construction. Similar to previous work [3, 9], we assume that the output p_i of an action a_i in a graph is consumed by a subsequent action, a_j. In other words, one of the K input objects $\{o_{jk}\}$ (including implicit entities)

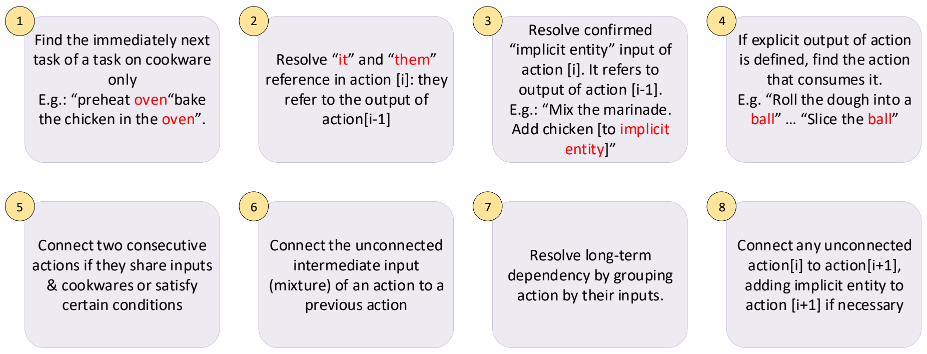

1
Find the immediately next task of a task on cookware only
E.g.: "preheat oven"bake the chicken in the oven".
2
Resolve "it" and "them" reference in action [i]: they refer to the output of action[i-1]
3
Resolve confirmed "implicit entity" input of action [i]. It refers to output of action [i-1].
E.g.: "Mix the marinade. Add chicken [to implicit entity]"
4
If explicit output of action is defined, find the action that consumes it.
E.g. "Roll the dough into a ball" ... "Slice the ball"
5
Connect two consecutive actions if they share inputs & cookwares or satisfy certain conditions
6
Connect the unconnected intermediate input (mixture) of an action to a previous action
7
Resolve long-term dependency by grouping action by their inputs.
8
Connect any unconnected action[i] to action[i+1], adding implicit entity to action [i+1] if necessary

Fig. 4: The rules used by our rule-based parser to convert procedure text into a flow-graph.

of a_j is equivalent to p_i, j indexes objects and k indexes actions. To connect the various text entities, we defined a set of rich semantic rules for the graph constructor. The full set of rules with examples is given in Figure 4. Connecting the various entities with these rules results in a complete fine-grained flow graph as shown in Figure 3. The fine-grained graph is then coarsened such that each node corresponds to a single instruction. The resulting coarsened graph is what serves as input to Graph2Vid.

4.2 Learning-based graph parsing

The adopted learning-based flow graph construction follows the same two steps of the rule-based approach described above. However, in this case both entity extraction and graph construction yield from learning-based neural networks trained on the English recipe flow graph corpus [9].
Text entity extraction. Here, the text entity extraction is the output of the tagger model used in [1]. In particular, this tagger is trained to recognize 10 different named entities as done in previous work [9]. For better accuracy, we re-trained the tagger on the Y-20 dataset [9]. We used the Adam optimizer with learning rate of 0.075 and a batch size of 30. We train the model for maximum of 100 epochs with early stopping using accuracy measure on the validation set as a metric, and patience period of 10 epochs. We evaluate the quality of the tagger by computing precision, recall and F1 score. For all 10 tags, we get precision, recall and F1 of 0.87, 0.88 and 0.87, respectively.
Flow graph construction. To construct the flow graph, we use the graph parser of [1], which takes as input the tagged entities from the previous step and converts them into a graph structure by predicting the presence of an edge between two entities, as well as a label indicating the semantic relation between them; see [9] for details on entity and edge sets used by the parser. Figure 5 shows an example of a recipe and (part of) its corresponding flow graph learned by our parser. We evaluate the parser using standard measures, such as Unlabeled

Attachment Score (UAS) and Labeled Attachment Score (LAS). UAS measures the number of nodes which are assigned correct parents, regardless of the edge label, while LAS takes into account correct edge label as well. Our parser yields UAS and LAS of 0.94 and 0.91, respectively.

4.3 Coarse flow graph generation

In both the rule-based and learning-based approaches, the obtained fine-grained flow graphs are collapsed into coarse sentence-level graphs to be used in our experiments. To go from fine-grained (i.e., entity-level) flow graphs to coarse (i.e., sentence-level) flow graphs, we traverse the fine-grained graphs using Depth First Search (DFS) and merge all nodes with same sentence ID into a single node, while retaining original connections.

5 Experimental

In this section, we detail the methods used for representation learning, presented in Table 2 of the main paper.

Feature extraction. We start by elaborating on the feature extraction procedure. Given a raw video Y and a tSort graph $\mathrm{S}_t = (V_t, E)$ where every node $t_i \in V_t$ contains a step description in the form of text, we apply jointly pre-trained video (f_v) and text (f_t) feature extractors [5] (as mentioned in Sec. 4.3 of the main paper) to convert the video into a sequence of clip embeddings, $X = f_v(Y)$ and the text in each node into a sentence embedding $v_i = f_t(t_i)$. All methods presented in Table 2 of the main paper expect such features as input.

Pre-trained Features. The baseline "Pre-trained Features" in Table 2 of the main paper does not use any training at all. Instead, we directly use the features from [5] (as described above) and perform flow-grounding in videos using Graph2Vid. In contrast to this method, all other approaches listed in Table 2 train a 2-layer Multi-Layer Perceptron (MLP) for the video features in order to improve step localization performance.

Bag of steps + soft clustering. The following baseline assumes that the the recipe is an unordered set of steps (thus disregarding the connections in $\mathcal{G}$), and uses soft clustering loss to improve the pre-trained representations. Given the sequence of video features, $X \in \mathbb{R}^{N \times d}$, and the list of step embeddings, $V \in \mathbb{R}^{K \times d}$, the soft clustering loss is defined as:

$$\mathcal{L}_{\text{clust}} = ||I - \hat{X}V^\top||_F, \tag{4}$$

where $I \in \mathbb{R}^{K \times K}$ is the identity matrix and $\hat{X} = (\hat{x}_1, \ldots, \hat{x}_K) \in \mathbb{R}^{K \times d}$. Each element $\hat{x}_i$ in $\hat{X}$ is defined according to

$$\hat{x}_i = \sum_{j=1}^{N} x_j \cdot \mathrm{softmax}(Xv_i/\gamma). \tag{5}$$

Recipe text

1. Heat oil in a pan and stir in ground beef. **Add** half of the chopped onions, chopped garlic and let **it** cook.
2. When the meat changes color and starts to release water, **add** chili powder and cumin powder. **Mix** and **cook** for 3 more minutes.
3. **Pour** in water and tomato paste, blend **it** well. Season with salt.
4. Thereafter mix in prepared yellow mustard, worcestershire sauce and brown sugar. Let it simmer for 5 minutes or until of desired thickness. Keep the chili aside.
5. Steam the frankfurters for 4 minutes. In the same steamer, place the hot dog buns and steam for a couple of minutes.
6. Place a frankfurter in the middle of the bun, cover the sausage with chill. Drizzle some mustard sauce over the top and garnish with chopped onions.

Recipe flow graph

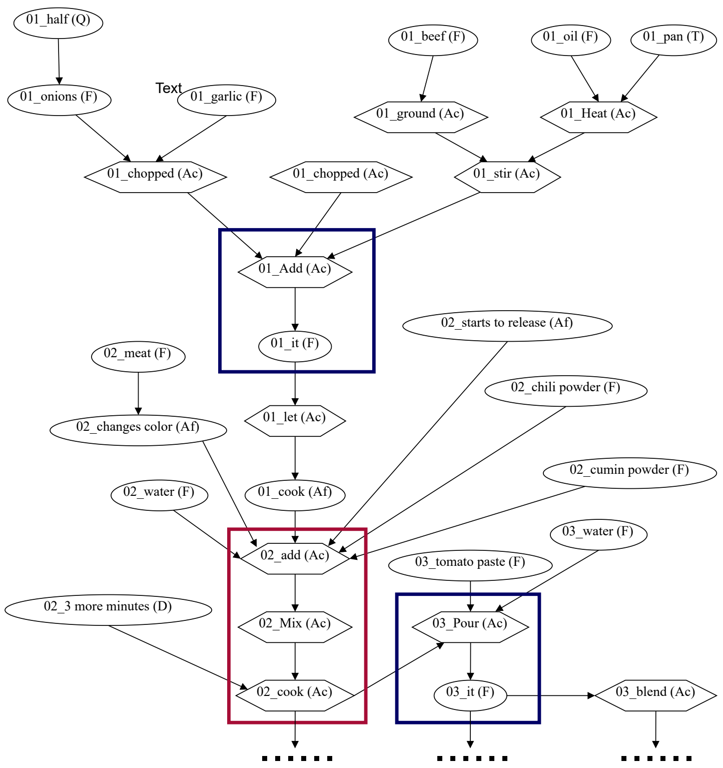


Fig. 5: A recipe from our dataset, along with part of its learned flow graph corresponding to steps 1–3. Action nodes (e.g., add, mix) are shown as hexagons, while other entities (e.g., Food, Tool) are shown as ovals. Each node is labelled by an id (corresponding to instruction step), a token, and its entity class/tag (e.g., F for Food, Ac for Action). Colored rectangles identify instances of co-reference and ellipsis resolution in our parser, and will be further discussed in Section 6.

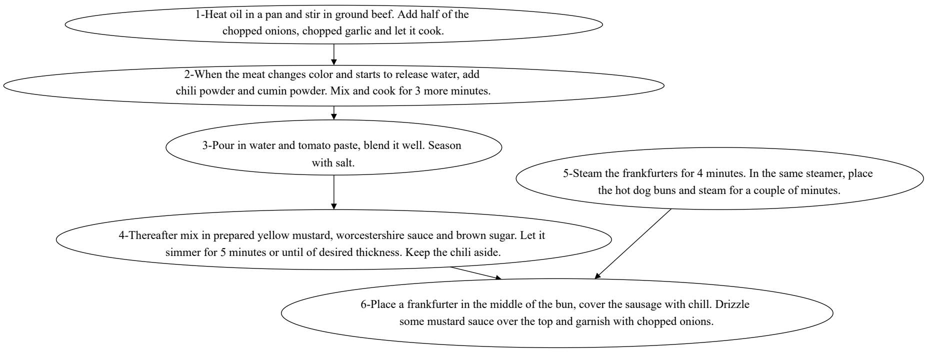

1-Heat oil in a pan and stir in ground beef. Add half of the chopped onions, chopped garlic and let it cook.
2-When the meat changes color and starts to release water, add chili powder and cumin powder. Mix and cook for 3 more minutes.
3-Pour in water and tomato paste, blend it well. Season with salt.
4-Thereafter mix in prepared yellow mustard, worcestershire sauce and brown sugar. Let it simmer for 5 minutes or until of desired thickness. Keep the chili aside.
5-Steam the frankfurters for 4 minutes. In the same steamer, place the hot dog buns and steam for a couple of minutes.
6-Place a frankfurter in the middle of the bun, cover the sausage with chill. Drizzle some mustard sauce over the top and garnish with chopped onions.

Fig. 6: A coarse flow graph for recipe in Figure 5. This is derived from a fine-grained recipe flow graph (part of it is shown in Figure 5).

In other words, $\hat{x}_i$ in Eq. (5) defines attention-based pooling of sequence x, relative to an element v_i. Minimizing $\mathcal{L}_{\text{clust}}$ pushes every element in V to have a unique match in X, which encourages the clustering of the embeddings x_i around the appropriate step embeddings v_i and thus promotes relevant feature learning. Note that this baseline does not use the knowledge of the flow graph structure or even the order of steps in V, since all the operations in soft clustering are permutation equivariant.

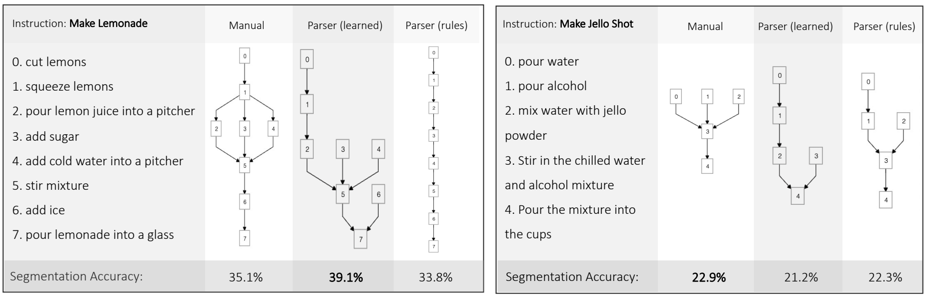
Linear Procedure + Drop-DTW. In the following baseline, we treat instructions, as listed in the procedural text, as an ordered list of steps. To train video features with step order supervision, we use Drop-DTW [2] and precisely follow their original implementation. The only difference between our baseline and the original Drop-DTW [2] is the source of supervision. Specifically, the original work uses the provided ground-truth steps order, while we use the steps of the generic procedure (identical for all the videos of the same category) in the order in which they appear in the procedure description.
To this end, given the video features $X \in \mathbb{R}^{N \times d}$, and the list of step embeddings, $V \in \mathbb{R}^{K \times d}$ we train the model with the following loss:

$$\mathcal{L}_{train}(V, X) = \mathcal{L}_{DTW}(V, X) + \mathcal{L}_{clust}(V, X), \tag{6}$$

Where $\mathcal{L}_{DTW}(X, V)$ is the cost of aligning the instruction list V with the video X, and $\mathcal{L}_{clust}$ is the soft clustering loss (defined above) that [2] proposes to add to regularize the training.

Graph2Vid. Finally, as described throughout our paper, Graph2Vid is a way to train video representations supervised by flow graphs, $\mathcal{G}$. Please refer to Section 3.4 of the main paper for a detailed description of this proposed formulation. Notably, to make the training with Graph2Vid more stable and avoid degenerate solutions where all the features map to a single graph node, we also adopt the regularization strategy from [2] and add the clustering loss. That is, our final training objective is

$$\mathcal{L}_{train}(G, X) = \mathcal{L}_{G}(G, X) + \mathcal{L}_{clust}(V, X), \tag{7}$$



Instruction: **Make Lemonade**	Manual	Parser (learned)	Parser (rules)
0. cut lemons 1. squeeze lemons 2. pour lemon juice into a pitcher 3. add sugar 4. add cold water into a pitcher 5. stir mixture 6. add ice 7. pour lemonade into a glass			
Segmentation Accuracy:	35.1%	**39.1%**	33.8%

Instruction: **Make Jello Shot**	Manual	Parser (learned)	Parser (rules)
0. pour water 1. pour alcohol 2. mix water with jello powder 3. Stir in the chilled water and alcohol mixture 4. Pour the mixture into the cups			
Segmentation Accuracy:	**22.9%**	21.2%	22.3%

Fig. 7: Manual vs parser-generated flow graphs. For two recipes, we provide the list of steps and derived from them flow graphs. The bottom row gives Graph2Vid segmentation accuracy on the videos of those procedures.

where $\mathcal{L}_G(G, X)$ is the cost of aligning the flow graph, $\mathcal{G}$, with the video, X, and $\mathcal{L}_{clust}$ is defined above in Eq (4).

Training details. In our experiment with Graph2Vid (as well as "Linear Procedure + Drop-DTW" and "Bag of steps + soft clustering" baselines) on CrossTask, we use a 2-layer MLP on top of the video features and train it with ADAM optimizer [4] with learning rate 10^{-4} and weight decay 10^{-4} for 10 epochs.

6 Flow Graph Analysis

In this section, we provide additional analysis of flow graphs and their influence on step localization performance.

6.1 Comparison of different flow graphs for step localization

To better understand the difference in step localization performance when using different flow graph, we compare manual and parser-generated flow graphs (see Fig. 7). On the "Make Lemonade" recipe (Fig. 7, left), the learning-based graph ignores some links present in the manual graph(e.g., $1 \rightarrow 3$, $1 \rightarrow 4$, and, $6 \rightarrow 7$) which leads to more parallel threads. On the other extreme, the rule-based graph detects a single linear order. This shows that the learning-based graph is more "flexible", which we argue is key for better segmentation. Inspecting the CrossTask dataset videos revealed that the sequence of steps $5 \rightarrow 6 \rightarrow 7$ (bottom of the manual graph) happens only 20% of the time in the data, while the alternative $5 \rightarrow 7 \leftarrow 6$ (from the learned parser) happens 80% of the time, which explains the higher learning-based segmentation accuracy (i.e., 39.1% vs 35.1%). Notably, sometimes both parsers produce more constrained flow graphs (e.g., Fig. 7, right), which leads to less accurate Graph2Vid segmentations.

6.2 Co-reference and ellipsis resolution by the learned parser

Here we provide an analysis of how our learned parser is capable of performing co-reference and ellipsis resolution. A quantitative analysis of the success rate of such resolutions requires a detailed manual annotation of the predicted flow graphs, and is beyond the scope of our study. But we observe many successful examples in our predicted flow graphs, which can be attributed to the way such resolutions are built into the annotation framework. Next, we present a few examples and elaborate on the reason behind the success of the parser in resolving them. Recall that the flow graph annotations of [9] draw on a set of pre-defined entity tags (including Action, Food, Tool, etc.), as well as a set of edge labels that identify relations among these entities (e.g., f-eq for food equivalency, t for target). These edge labels are used to connect nodes of different entity types. E.g., when f-eq connects two Food nodes, it signifies that the two Food items are equivalent (e.g., *potato* and *diced potato*). This same label can be used to connect an Action and a Food node, meaning that the result of the Action is the same as the Food node.

The edge labels by design can help resolve the referent of pronouns that refer back to the result of a previous cooking action. We can see two such examples in the flow graph shown in Figure 5 (inside the blue boxes), which result in correct resolution of references for the pronoun *it*. Similarly, the edge label t can connect an Action to its direct object (e.g., the connection from *heat* (Ac) to *oil* (F) in Figure 5), but also an Action to another Action whose result is the missing direct object of the first Action, and as such can help with ellipsis resolution. Figure 5 provides two examples of such ellipsis resolution (inside the red box), where the result of *add* is identified as the implicit (missing) argument of *mix*, whose result is in turn linked to *cook* as its (missing) direct object. All in all, because the training data contains many instances where co-reference and ellipsis cases are explicitly resolved via the use of proper edge connections and labels among entities, the parser is in principle capable of resolving them at inference time.

References

1. Donatelli, L., Schmidt, T., Biswas, D., Köhn, A., Zhai, F., Koller, A.: Aligning actions across recipe graphs. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (2021)
2. Dvornik, N., Hadji, I., Derpanis, K.G., Garg, A., Jepson, A.: Drop-DTW: Aligning common signal between sequences while dropping outliers. In: Advances in Neural Information Processing Systems (NeurIPS) (2021)
3. Kiddon, C., Ponnuraj, G.T., Zettlemoyer, L., Choi, Y.: Mise en place: Unsupervised interpretation of instructional recipes. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP) (2015)
4. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
5. Miech, A., Alayrac, J.B., Smaira, L., Laptev, I., Sivic, J., Zisserman, A.: End-to-End Learning of Visual Representations from Uncurated Instructional Videos. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2020)
6. Schumacher, P., Minor, M., Walter, K., Bergmann, R.: Extraction of procedural knowledge from the web: A comparison of two workflow extraction approaches. In: Proceedings of the 21st International Conference on World Wide Web (2012)
7. spaCy: https://spacy.io/
8. Weibel, C.A.: An introduction to homological algebra. No. 38, Cambridge university press (1995)
9. Yamakata, Y., Mori, S., Carroll, J.: English recipe flow graph corpus. In: Proceedings of the 12th Language Resources and Evaluation Conference (2020)