In this short note, we include several implementation and algorithmic details about our decoders (Section 1) and our neural network models (Section 2), as well as additional information about the datasets and the component representations used (Section 3).

1 Shortest Path Decoders

1.1 $k$-Shortest Paths

In our paper, we implement Yen’s algorithm (Yen, 1971) to compute $k$-SSSPs when doing decoding. For completeness, we show the full algorithm in Algorithm 1. As detailed in Brander and Sinclair (1995), this method is one of many $k$ Shortest Path algorithms that works by finding deviation or branching paths from an initial SSSP (computed in Line 3). For each $k$ starting on Line 4, the method then dissects the most recent shortest path and again uses the single shortest path method to find an alternative path from each point new node that hasn’t been observed in the current list $A$ (as checked starting on line 8).

In the case of DAGs, this algorithm has a time complexity of $O(k|\mathcal{G}|^2)$ for $k > 1$. This complexity can be explained in the following way: $O(|\mathcal{G}|)$ (where $|\mathcal{G}|$ is short hand for the the size of the graph, or $|V| + |N|$) is the complexity of the DAG single shortest path procedure (or for $k = 1$). For each $k$, we consider $l$ number of new start positions in the most recent $k − 1$ SSSP (starting on Line 5), which in the worst case can be of size $|\mathcal{G}|$. Each branching path $j \in l$ then requires a run of the SSSP procedure of complexity $O(|\mathcal{G}|)$ (as stated above), thus involving $|\mathcal{G}|^2$ amount of possible searching for each $k$.

In Algorithm 1, we show several optimizations that improve the runtime (though not the complexity) of the procedure, including starting each nested call to SP at new start as opposed to searching through the full graph, and using a min heap to store candidate shortest path in Lines 2,13 and 15 as opposed to having to re-sort $B$ each time at line 15. Another frequently used optimization trick (not shown here), known as Lawler’s trick (Lawler, 1972), involves keeping track of already computed branching paths so as to avoid solving for duplicate candidates shortest path in $B$ and having to make repeated calls to the SP procedure (line 11). This last trick significantly improved the running time of our decoders (see Brander and Sinclair (1995) for more details and analysis).

1.2 Lexical Shortest Path Implementation

An illustration of the lexical SSSP search is shown in Figure 1, which highlights several implementation details described below.

Due to this approximation, our decoder as implemented and described above is not exact, as proved by the counter example shown in Figure 2. In general, since the normalizer is computed at the terminating node (as opposed to during the SSSP search), longer sequences can block shorter sequences with higher (post normalized) probability.
Table 1 shows the different neural network settings across the different datasets.

Despite this, we found this method to be empirically optimal for \( k > 1 \) when compared against our previous work (Richardson and Kuhn, 2017) (in which an exact, albeit less efficient, method is used). An additional implementation trick is that after each candidate SSSP is found (line 12 in Algorithm 1), we run our translation model on the input and full candidate again to compute the correction score.

1.3 Longest vs. Shortest Paths

When working with DAGs, we could also solve for longest paths by replacing \( \min \) with \( \max \) in Equation 3. We use \( \min \) since our method will work equally well for other types of graph path problems where using \( \max \) is not feasible.

On this last point, it is important to note that while our experiments deal exclusively with DAGs, which is motivated by the simplicity of the target component languages, more complex graphs could be used in our framework by simply replacing our SSSP method with SSSP methods that are suited to such graphs.

2 Neural Models

2.1 Hyper Parameters

Table 1 shows the different neural network settings across our datasets. In all datasets, we used shallow networks with a single layer encoder and decoder, and early stopping by monitoring training progress to a validation set. Similarly, we used vanilla stochastic gradient descent in all cases and did not use any form of regularization or drop out, since this did not seem to help. Standardly, we normalize the resulting log probability of candidate translations by the length of the target sentence. All models were implemented using the Cython wrapper for DyNet (Neubig et al., 2017).

In each case, we modeled out of vocabulary by mapping each training token with a frequency of 2 or less to an artificial OOV token.

3 Dataset Credits and Details

3.1 Credits and New Data

The additional Japanese Python and Lua datasets were taken from the following resources (respectively): http://docs.python.jp/2/ and https://www.lua.org/manual/. All datasets are publicly available (see below). We note that the Java datasets was first investigated in Deng and Chrupala (2014), and the Unix dataset was first introduced in Richardson and Kuhn (2014).

3.2 Sportscaster

When working with the Sportscaster corpus, one issue is that each training item is paired not with a gold meaning representation, but a set of possible meaning representations, and as such involves learning with ambiguous supervision. One idea, which we pursued early on, is to train our translation and neural models on all possible pairs, including incorrect pairs, which lead to sub-optimal

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Epochs</th>
<th>Embedding</th>
<th># Hidden States</th>
<th>beam</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sportscaster</td>
<td>25 (max.)</td>
<td>200</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>GeoQuery</td>
<td>20 (max.)</td>
<td>250-300</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>Tech. Docs.</td>
<td>8 (max.)</td>
<td>250-300</td>
<td>100-150</td>
<td>2-3</td>
</tr>
</tbody>
</table>

Algorithm 1 k-SSSP Decoding via Yen’s Algorithm

**Input:** Input x, DAG G, SSSP method SP, number of paths K, translation mode \( \theta \), starting node b.

**Output:** \( k \) shortest paths \( A \)

1: \( A[k] \leftarrow Nil \)
2: \( B \leftarrow [\] \)
3: \( A[0] \leftarrow SP(x, G, \theta, b) \)
4: for \( k \in 1..K \) do
5:     for \( i \in 0 \) to \( \text{len}(A[k-1]) \) do
6:         new_start \( \leftarrow A[k-1][i] \)
7:         root \( \leftarrow A[k-1][i] \)
8:     for each path \( p \in A \) do
9:         if root \( = p[0 : i] \) then
10:             \( G \leftarrow \text{Block}(G, p[i], p[i+1]) \)
11:         branching \( \leftarrow \text{SP}(x, G, \theta, \text{new_start}) \)
12:         candidate \( \leftarrow \text{root} + \text{branching} \)
13:         \( B \leftarrow \text{HEAPPush}(B, \text{candidate}) \)
14:         \( G \leftarrow \text{UNBLOCK}(G) \)
15:     \( A[k] \leftarrow \text{HEAPPOP}(B) \)
16: return \( A \)

\( \triangleright \) Initialize the k-best list \( A \)
\( \triangleright \) Initialize the k-best candidate list \( B \)
\( \triangleright \) Find initial SSSP starting from b
\( \triangleright \) Run through each node in recent SSSSP
\( \triangleright \) Find all paths in \( A \) matching root and block next point
\( \triangleright \) Find new SSSP from new_start
\( \triangleright \) Add candidate as a candidate shortest path
\( \triangleright \) Add best candidate to \( A \)
Graph State

Search Position and Score Computation

Initialize $d \leftarrow \infty$, $s \leftarrow [0.0, 0.0, 0.0, 0.0]$

Node: 0

$s_0 = \{p_1(x_1 \mid \lambda), p_1(x_2 \mid \lambda), p_2(x_3 \mid \lambda), p_3(x_4 \mid \lambda)\}$

Node: 1

$\text{score}_{0 \rightarrow 1} = -\log \prod (s_0 + [p_1(x_1 \mid f_{\text{fun}_1}), p_1(x_2 \mid f_{\text{fun}_1}), p_1(x_3 \mid f_{\text{fun}_1}), p_3(x_4 \mid f_{\text{fun}_1}])$}

Node: 2

$\text{score}_{0 \rightarrow 2} = -\log \prod (s_0 + [p_1(x_1 \mid f_{\text{fun}_2}), p_1(x_2 \mid f_{\text{fun}_3}), p_1(x_3 \mid f_{\text{fun}_4}), p_3(x_4 \mid f_{\text{fun}_2}])$

Node: 3

$\text{score}_{2 \rightarrow 3} = -\log \prod (s_2 \rightarrow [p_1(x_1 \mid x), p_1(x_2 \mid x), p_1(x_3 \mid x), p_2(x_4 \mid x)]$}

Node: 4

$\text{score}_{3 \rightarrow 4} = -\log \prod (s_3 \rightarrow [p_1(x_1 \mid y), p_1(x_2 \mid y), p_1(x_3 \mid y), p_3(x_4 \mid y)]$}

Node: 5

$\text{score}_{4 \rightarrow 5} = -\log \prod (s_4)$

Back traversal to source node 0 and path discovery

<table>
<thead>
<tr>
<th>function_1</th>
<th>function_2</th>
<th>$p(x_1)$</th>
<th>$p(x_2)$</th>
<th>$p(x_3)$</th>
<th>$p(x_4)$</th>
<th>$p(x \mid \lambda)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{\text{fun}_1}$</td>
<td>$f_{\text{fun}_2}$</td>
<td>0.6</td>
<td>0.1</td>
<td>0.1</td>
<td>0.05</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Figure 1: An illustration of the lexical SSSP algorithm for the text input $x = \text{function}_1 \text{ applied to arg}_x$. The table for $p_x$ is on the bottom, where $\lambda$ denotes an artificial NULL word token on the target side.
Figure 2: An example graph and decoding run where the lexical SSSP search does not find the correct 1-best translation (involving the excluded red edge) of the input function_1 (uses $p_t$ from Figure 1).

results. Instead, we disambiguated the data by training an initial word translation model on this ambiguous dataset, then used this model to disambiguate and select the most probable meaning representation. All other models were then trained on this disambiguated dataset. In doing this, we follow the original experiments by Chen and Mooney (2008).

### 3.3 Component Representations

As discussed in the paper, the component languages are finite languages, since each API contains only a finite number of defined or valid functions. As such, not only is this task a constrained machine translation problem, but also a constrained semantic parsing problem. See (Richardson and Kuhn, 2017) for more discussion about this and a comparison with the related task of automatic algebra word problem solving (where a similar assumption is often made about the set of valid algebra equation templates being finite).

For more details about the target component representations, including a new way of normalizing these representations across programming languages and translating these normalized forms into classical logic, see Richardson (2018).

### 4 Resources


### References


