Noise reduction and targeted exploration in imitation learning for Abstract Meaning Representation parsing

James Goodman* Andreas Vlachos† Jason Naradowsky*
* Computer Science Department, University College London
james@janigo.co.uk, jason.narad@gmail.com
† Department of Computer Science, University of Sheffield
a.vlachos@sheffield.ac.uk

A Supplemental Material

Here we provide a detailed system description1. There are two scala packages. The dagger package contains the core imitation learning algorithm implementation, and dagger-amr package contains the implementation of the transition system for AMR parsing. dagger-amr is dependent on dagger.

A.1 AMR Fragments

Flanigan et al. (2014) and Wang et al. (2015b), both use AMR fragments as their smallest unit, which may consist of more than one AMR concept. Instead we work with the individual AMR nodes, and rely on Insert actions to learn how to build common fragments, such as country names. The main adaptations to the actions stem from this. Wang et al. (2015a) later introduced an ‘Infer’ action similar to our Insert action. Infer inserts an AMR concept node above the current node as Insert does, but is restricted to nodes that occur outside of AMR ‘fragments’, which continue to be the base building block. Their Merge action merges $\sigma_0$ and $\beta_0$ into a composite node; this is not required with the retention of a 1:1 mapping between nodes and AMR concept. Figure 1 shows an example for the fragment that represents “NATO”. The internal structure to this fragment is invisible to during learning or execution in Flanigan et al. (2014; Wang et al., 2015b).

A.2 Action Space

Figure 2 shows a parse of a sentence fragment. The current $\sigma_0$ node is shown dashed and in red. This illustrates that we stay in a uniform graph-space throughout with the input dependency tree

---

1Code available at https://github.com/hopshackle/dagger-AMR.
<table>
<thead>
<tr>
<th>Action Name</th>
<th>Param.</th>
<th>Pre-conditions</th>
<th>Outcome of action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NextEdge</td>
<td>( l_r )</td>
<td>( \beta ) non-empty</td>
<td>Set label of edge ((\sigma_0, \beta_0)) to relation ( l_r ). Pop ( \beta_0 ).</td>
</tr>
<tr>
<td>NextNode</td>
<td>( l_c )</td>
<td>( \beta ) empty</td>
<td>Set concept of node ( \sigma_0 ) to concept ( l_c ). Pop ( \sigma_0 ), and re-initialise ( \beta ).</td>
</tr>
<tr>
<td>Swap</td>
<td>( \beta ) non-empty</td>
<td>Make ( \beta_0 ) parent of ( \sigma_0 ) (reverse edge) and its sub-graph. Pop ( \beta_0 ) and insert ( \beta_0 ) as ( \sigma_1 ).</td>
<td></td>
</tr>
<tr>
<td>ReplaceHead</td>
<td>( \beta ) non-empty</td>
<td>Pop ( \sigma_0 ) and delete it from the graph. Parents of ( \sigma_0 ) become parents of ( \beta_0 ). Other children of ( \sigma_0 ) become children of ( \beta_0 ). Insert ( \beta_0 ) at the head of ( \sigma ) and re-initialise ( \beta ).</td>
<td></td>
</tr>
<tr>
<td>Reattach</td>
<td>( \kappa )</td>
<td>( \beta ) non-empty</td>
<td>Pop ( \beta_0 ) and delete edge ((\sigma_0, \beta_0)), and attach ( \beta_0 ) as a child of the node ( \kappa ). If ( \kappa ) has already been popped from ( \sigma ) then re-insert it as ( \sigma_1 ).</td>
</tr>
<tr>
<td>DeleteNode</td>
<td>( \beta ) empty; ( \sigma_0 ) is leaf node</td>
<td>Pop ( \sigma_0 ) and delete it from the graph.</td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td>( l_c )</td>
<td>( \sigma_0 ) is a leaf node</td>
<td>Insert a new node ( \delta ) with AMR concept ( l_c ) as the parent of ( \sigma_0 ), and insert ( \delta ) into ( \sigma ).</td>
</tr>
<tr>
<td>InsertBelow</td>
<td>( l_c )</td>
<td>( \sigma_0 ) is a leaf node</td>
<td>Inserts a new node ( \delta ) with AMR concept ( l_c ) as the child of ( \sigma_0 ).</td>
</tr>
</tbody>
</table>

Table 1: Action Space for the transition-based graph parsing algorithm

incrementally changes to the output AMR graph. From the top the actions are

- Insert(date-entity)
- NextNode(WORD)
- NextEdge(year)
- second diagram
- NextNode(WORD)
- ReplaceHead to remove “in”
- third diagram
- NextNode(WORD)
- NextEdge(mod)
- Reattach to move “date-entity”
- fourth diagram
- NextNode(VERB)
- ReplaceHead to remove “by”
- NextEdge(ARG0)
- NextEdge(time)
- NextNode(strike-01)

The actions in the space are summarised in Table 1, with details in the following sections.

A.3 NextNode, NextEdge and Delete

NextNode and NextEdge form the core of the algorithm. We progress over all nodes from the bottom of the tree up, first labelling the outgoing edges with an AMR relation using NextEdge, and then labelling the node with an AMR concept with NextNode before moving to the next node in the \( \sigma \) stack. Figure 3 shows an example of NextNode and NextEdge actions, and these are unchanged from Wang et al. (2015b). Each NextNode action is parameterised with \( l_c \), the AMR concept to be used as the label, and each NextEdge action is parameterised with \( l_r \), the AMR relation to be used.

Delete removes a leaf node completely from the graph where the word does not map to any AMR concept. An example is included in Figure 3.

Figure 3: The three graphs show parts of successive states starting from a dependency tree (on the left). The actions from left to right are to Delete “The”; Label the “center” node as center with NextNode.

Figure 4: The two graphs continue from Figure 3. The actions are to label the “nsubj” edge as ARG0 with NextEdge; Label the “bolster” node as bolster-01 with NextNode.
A.4 Swap, Reattach and ReplaceHead

These actions change the overall structure of the graph, but always retain a tree structure provided they start from a tree.

- Swap reverses the direction of an edge, so that $\beta_0$ becomes the parent of $\sigma_0$ and its sub-graph. An example is shown in Figure 5.
- Reattach takes the sub-graph starting at $\beta_0$, detaches it from $\sigma_0$ and moves it to a new parent $\kappa$. An example is shown in Figure 6.
- ReplaceHead removes a node from the graph that is not a leaf node (in which case the Delete action would be used). An example is shown in Figure 7.

Unlike Wang et al. (2015b) we do not parameterise Swap or Reattach actions with a relation label. We leave that decision to a later NextEdge action. We permit a Reattach action to use parameter $\kappa$ equal to any node within a distance of six edges from $\sigma_0$, excluding any node in the sub-graph of $\beta_0$ to avoid disconnecting the graph and creating loops. This is slightly more than Wang et al. use, and we found the increase helpful to cope with the larger graphs we have given the avoidance of ‘fragments’ that merge multiple AMR Concept nodes in the parsed graph.

Our ReplaceHead covers two distinct actions in Wang et al. (2015b); ReplaceHead and Merge. The Merge action merges $\sigma_0$ and $\beta_0$ into a composite node, that keeps all the words represented in the final AMR graph. This is not required in our approach as we do not have composite nodes and retain a 1:1 mapping between nodes and AMR concept.

A.5 Insert and InsertBelow

The Insert/InsertBelow actions insert new node as a parent/child of the current $\sigma_0$. The action is parameterised with $l_c$, the AMR concept for the inserted node. Neither action exists in Wang et al. (2015b), while Wang et al. (2015a) introduces an ‘Infer’ action that is equivalent to Insert. When a node is inserted, we set the lemma equal to the AMR concept, to be used in features for future actions. An example is shown in Figure 8.
A.6 Reentrance

Wang et al. (2015b) have a Reentrance action that creates a new edge between the current node $\sigma_0$, and another nearby node $\kappa$. Since none of the other actions will convert a tree into a non-tree, the exclusion of Reentrance means that the output AMR graph is always a tree. Despite this strong restriction, we found the accuracy of results was better if Reentrance was excluded.

A.7 Additional action constraints

In our approach $T$ is theoretically unbounded and the algorithm could Insert, or Reattach ad infinitum.

We impose constraints to prevent these situations. Specifically:

- A Swap action cannot be applied to a previously Swapped edge
- Once a node has been moved by Reattach, then it cannot be Reattached again
- An Insert action can only be executed once with any given node as $\sigma_0$
- An Insert action is not permissible if it would insert an AMR concept that is already in use as any of the parent, children, grand-parents or grand-children of $\sigma_0$

We only allow actions which preserve acyclicity, but do not prevent duplication of argument relations so that a concept could have two outgoing $\text{ARG1}$ edges even if this is impossible linguistically. We start with a fully connected graph (the dependency tree), and preserve full connectivity as none of the actions will disconnect a graph.

A.8 The Expert Policy

The expert policy used in training applies heuristic rules to determine the next action from a given state. It uses the training alignments to construct a mapping between nodes in the dependency tree, and nodes in the target AMR. Any unmapped nodes in the dependency tree will be deleted by the expert, and any unmapped nodes in the AMR graph will be inserted. The JAMR aligner maps a sequence of words to an AMR graph fragment, so there is an additional alignment stage local to the expert that maps individual nodes in the AMR fragment to words in the sequence. This is done by calculating the Jaro string distance (Jaro, 1989) between each pairing of AMR concept and word, and then greedily assigning pairs starting with the best match.

From any given state, the expert takes an action from the following rules listed in priority order. In these rules, ‘AMR’ refers to the Gold AMR graph that the expert is aiming to produce. ‘Current’ refers to the state of the graph during processing.

1. If both $\sigma_0$ and $\beta_0$ map to AMR nodes, and there is an AMR relation from $\beta_0$ to any ancestor of $\sigma_0$ then apply Swap to reverse the arc and put the old $\beta_0$ node above $\sigma_0$
2. If the current node, $\sigma_0$, is mapped to an AMR

parameters ($l_c$ and $l_r$) if they appear in any sentence containing the same lemma as $\sigma_0$ and $\beta_0$. We reduce this to just concepts that have been aligned to the current lemma in the training data.

We initially run the expert policy over the training set, and track the AMR concept assigned for each lemma. These provide the possible $l_c$ that will be used for NextNode actions. Similarly we track the lemmas at head and tail of each expert-assigned AMR relation, and compile possible $l_r$ from these.

This means that AMR concepts/relations will never be considered during test if they were not aligned to that lemma in the training data. To relax this restriction we also allow $l_c$ to take the values \text{WORD, LEMMA, VERB}, which respectively use the actual word, lemma, or the lemma concatenated with ‘-01’ as the AMR concept. This is inspired by Werling et al. (2015), who use a similar set of actions in a concept identification phase. For the $l_c$ parameters on Insert (InsertBelow) actions, we use all AMR concepts that the expert inserted above (below) any node in the training set with the same lemma as $\sigma_0$.

The expert policy used in training applies a number of heuristics to determine the next action from a given state. It uses the alignments from the JAMR aligner to construct a mapping between nodes in the starting dependency tree, and nodes in the target AMR graph. Any unmapped nodes in the dependency tree will then need to be deleted by the expert, and any unmapped nodes in the AMR graph will need to be inserted. The JAMR aligner maps a sequence of words to an AMR graph fragment, so there is an additional alignment stage local to the expert that maps individual nodes in the AMR fragment to words in the sequence. This is done by calculating the Jaro string distance (Jaro, 1989) between each pairing of AMR concept and word, and then greedily assigning pairs starting with the best match.

From any given state, the expert takes an action from the following rules listed in priority order. In these rules, ‘AMR’ refers to the Gold AMR graph that the expert is aiming to produce. ‘Current’ refers to the state of the graph during processing.

1. If both $\sigma_0$ and $\beta_0$ map to AMR nodes, and there is an AMR relation from $\beta_0$ to any ancestor of $\sigma_0$ then apply Swap to reverse the arc and put the old $\beta_0$ node above $\sigma_0$
2. If the current node, $\sigma_0$, is mapped to an AMR
node, and this AMR node has an unmapped AMR node as parent, then Insert a new node and map this to the unmapped parent AMR node.

3. If $\sigma_0$ is mapped to an AMR node, and this node has a child leaf node that is not yet mapped to any node in the graph, then InsertBelow to create this node, and update the mapping.

4. If $\beta$ is empty (i.e. all outgoing edges from $\sigma_0$ have already been labelled), and $\sigma_0$ has a mapping to an AMR node, then label $\sigma_0$ with the appropriate concept using NextNode.

5. If $\sigma_0$ is a leaf node and has no mapping to an AMR node, then Delete it.

6. If $\sigma_0$ has no mapping to an AMR node, but $\beta_0$ does, then apply ReplaceHead to merge $\sigma_0$ into $\beta_0$.

7. If both $\sigma_0$ and $\beta_0$ map to AMR nodes, and there is an AMR relation $\sigma_0 \rightarrow \beta_0$, then label with the appropriate relation using NextEdge.

8. If both $\sigma_0$ and $\beta_0$ map to AMR nodes, and there is an AMR relation $\beta_0 \rightarrow \sigma_0$ then apply Swap to reverse this arc.

9. If $\beta_0$ is mapped to an AMR node with a parent that is not mapped to $\sigma_0$, then Reattach $\beta_0$ to the correct node according to the mapping.

10. If $\beta$ is not empty, then apply NextEdge to label the relation using the current label on the edge $(\sigma_0 \rightarrow \beta_0)$.

11. Use NextNode to label the node using an AMR concept equal to the word of the node.

The last two actions ensure an action is always possible and may result in relations and concepts in the final AMR graph that are not in the AMR vocabulary. For example an edge might be labelled $\text{nsubj}$ from the starting dependency label. This will simply reduce the final F-Score.

The expert and the AMR aligner obtain an F-Score of 0.94 on the training data.

### A.9 Features

Features used are detailed in Tables 2 and 3. All are 0-1 indicator functions. Comparator and Negation features are inspired by some of the JAMR pre-processing steps (Flanigan et al., 2014).

The key differences to Wang et al. (2015b) are the inclusion of the brown, POSpath, NERpath, prefix and suffix feature types. Wang et al. (2015a) does include Brown cluster information in the feature set, as well as other features from a semantic

<table>
<thead>
<tr>
<th>Context</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_0$</td>
<td>lemma, comparator, negation, dl, ner, POS, inserted, prefix, suffix, brown, deleted, lemma-dl</td>
</tr>
<tr>
<td>$\sigma_0P$</td>
<td>inserted, lemma, brown</td>
</tr>
<tr>
<td>$\sigma_0C$</td>
<td>label, ner, label-brown</td>
</tr>
<tr>
<td>$\beta_0$</td>
<td>inserted, POS, lemma, brown, ner, dl, prefix, suffix, merged</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>ner, POS, lemma, brown, label</td>
</tr>
<tr>
<td>$\sigma_0 \rightarrow \beta_0$</td>
<td>label, path, lemma-path-lemma, POSpath, inserted-inserted, lemma-POS, POS-lemma, dl-lemma, lemma-dl, lemma-label, label-lemma, ner-ner, distance</td>
</tr>
<tr>
<td>$\beta_0 \rightarrow \kappa$</td>
<td>path, lemma-path-lemma, NERpath, POSpath, distance, lemma-POS, dl-lemma, ner-ner</td>
</tr>
<tr>
<td>$\sigma_0 \rightarrow \kappa$</td>
<td>distance, lemma-path-lemma, brown-brown, NERpath, POSpath, lemma-dl, lemma-label</td>
</tr>
<tr>
<td>$\sigma_0P \rightarrow \sigma_0$</td>
<td>label, POS-lemma, dl-lemma, ner-ner</td>
</tr>
<tr>
<td>$\sigma_0P \rightarrow \sigma_0$</td>
<td>lemma-lemma-lemma</td>
</tr>
<tr>
<td>$\sigma_0 \rightarrow \sigma_0C$</td>
<td>POS-lemma, lemma-POS, dl-lemma, ner-ner</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparator</td>
<td>word terminates in ‘er’ or ‘est’</td>
</tr>
<tr>
<td>negation</td>
<td>word starts with ‘un’, ‘in’, ‘il’ or ‘anti’</td>
</tr>
<tr>
<td>inserted</td>
<td>node was inserted by the parser</td>
</tr>
<tr>
<td>dl</td>
<td>dependency label in the original dependency tree</td>
</tr>
<tr>
<td>ner</td>
<td>named entity tag from Stanford parser in pre-processing</td>
</tr>
<tr>
<td>POS</td>
<td>part-of-speech tag from pre-processing</td>
</tr>
<tr>
<td>prefix</td>
<td>string before hyphen if word is hyphenated</td>
</tr>
<tr>
<td>suffix</td>
<td>string after hyphen if word is hyphenated</td>
</tr>
<tr>
<td>brown</td>
<td>cuts at 4, 6, 10 and 20 from 320-class Brown Clusters</td>
</tr>
<tr>
<td>deleted</td>
<td>lemma of any child node previously deleted by the parser</td>
</tr>
<tr>
<td>merged</td>
<td>lemma of any node merged into this node by a ReplaceHead action</td>
</tr>
<tr>
<td>distance</td>
<td>distance between the tokens in the sentence</td>
</tr>
<tr>
<td>path</td>
<td>concatenation of lemmas and dl’s between the tokens in the starting dependency tree</td>
</tr>
<tr>
<td>POSpath</td>
<td>concatenation of POS tags between the tokens in the starting dependency tree</td>
</tr>
<tr>
<td>NERpath</td>
<td>concatenation of NER tags between the tokens in the starting dependency tree</td>
</tr>
</tbody>
</table>
roll labeller and co-reference resolver.

A.10 Pre-processing and initialisation

We undertake some pre-processing on the English sentences, primarily to deal efficiently with dates and numbers. The pre-processing steps are:

- Insert spaces around any '/' characters to ensure strings like 'and/or' are tokenised as two words. Hyphenated words left as single tokens.
- Pass the full sentence through the Stanford Dependency Parser to construct a dependency tree (Manning et al., 2014), including annotation on parts-of-speech, named entity recognition, lemmas and dependency labels (all used as Features in Section 2). We use v3.3.1 of the Stanford Parser.
- Any tokens representing punctuation marks are then removed. During cross-validation, it was determined that leaving punctuation marks in the starting dependency tree did not improve performance.
- Match any month name (‘January’, ‘Jan’, ‘March’ etc.) and replace it with the month number mm.
- Match any numeric strings of format ddmmyy or dd-mm-yyyy and change these to dd mm yyyy to provide a set of three numeric tokens from which the AMR date-entity structure can be learned.
- Match any number string between ‘one’ and ‘twelve’ and replace it with the relevant numeric digits.
- Match any string of ‘hundred’, ‘thousand’, ‘million’ or ‘billion’ immediately preceded by a number, and replace both word tokens with the numeric amount - e.g. “2 thousand” becomes “2000”.

In AMR the convention is that any amount is expressed in digits, regardless of the form in the text, and this pre-processing enables these amounts to be used directly in the AMR graph.

A.11 Command line

For the DA GGER experiments reported in section 6 of the main paper, the command line used was:

    java -Xms8g -Xmx12g -jar ../amr-dagger-J7.jar
    --dagger.output.path ./
    --dagger.iterations 10 --train.data
    ../amr-1.0-proxy-train.txt
    --validation.data
    ../amr-1.0-proxy-dev.txt
    --algorithm Dagger
    --initialExpertProb 1.0
    --oracleLoss true
    --maxActions 300
    --aligner JAMR
    --classifier AROW
    --arow.smoothing 100
    --WXfeatures PCKX
    --reentrance false
    --reducedActions false
    --punctuation false
    --arow.iterations 5
    --preferKnown false
    --fileCache true
    --instanceThreshold 10
    --minTrainingSize 120
    --maxTrainingSize 120
    --startingClassifier false
    --wikification false
    --average true
    --previousTrainingIter 2
    --logTrainingStats false
    --brownCluster ../Brown320.txt
    --textEncoding UTF-8

The parameters --classifier, --arow.smoothing, --instanceThreshold were varied, with respective values of {AROW, PA, PERCEPTRON}, {10, 100, 1000}, {1, 100}. --instanceThreshold controls the α-bound.

For the v-DAGGER experiments with targeted exploration reported in section 6 of the main paper, the command line used was:

    java -Xms8g -Xmx12g -jar ../amr-dagger-J7.jar
    --dagger.output.path ./
    --dagger.iterations 10 --train.data
    ../amr-1.0-proxy-train.txt
    --validation.data
    ../amr-1.0-proxy-dev.txt
    --algorithm Dagger
    --initialExpertProb 1.0
    --oracleLoss false
    --maxActions 300
    --aligner JAMR
    --arow.smoothing 1000
    --WXfeatures PCKX
    --reentrance false
    --reducedActions true
    --punctuation false
    --arow.iterations 5
    --preferKnown false
    --fileCache true
    --instanceThreshold 1
    --threshold 0.10
    --minTrainingSize 120
    --maxTrainingSize 120
    --startingClassifier false
    --wikification false
    --average true
    --previousTrainingIter 2
    --rolloutLimit 10
    --logTrainingStats false
    --brownCluster ../Brown320.txt
    --textEncoding UTF-8
The parameter --threshold was varied, with respective values of \{0.02, 0.05, 0.10, 0.20\}. Focused costing is switched on with the additional parameter setting --expertHorizon true. The parameters for focused costing are then set with, using the 5/5 setting as an example, --expertAfter 5 --expertHorizonInc 5.

References

Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A Smith. 2014. A discriminative graph-based parser for the abstract meaning representation.


