A Transition-Based Directed Acyclic Graph Parser for UCCA
Supplementary Notes

A Feature Templates

Figure 1 presents the feature templates used by $\text{TUPA}_{\text{Sparse}}$. All feature templates define binary features. The other classifiers use the same elements listed in the feature templates, but all categorical features are replaced by vector embeddings, and all count-based features are replaced by their numeric value.

For some of the features, we used the notion of *head word*, defined by the h^* function (see Appendix D). While head words are not explicitly represented in the UCCA scheme, these features prove useful as means of encoding word-to-word relations.

Features from (Zhang and Clark, 2009):

unigrams

s_0tde, s_0we, s_1tde, s_1we, s_2tde, s_2we, s_3tde, s_3we,
b_0wtd, b_1wtd, b_2wtd, b_3wtd,
s_0lwe, s_0rwe, s_0uwe, s_1lwe, s_1rwe, s_1uwe

bigrams

s_0ws_1w, s_0ws_1e, s_0es_1w, s_0es_1e, s_0wb_0w, s_0wb_0td,
s_0eb_0w, s_0eb_0td, s_1wb_0w, s_1wb_0td, s_1eb_0w, s_1eb_0td,
b_0wb_1w, b_0wb_1td, b_0tdb_1w, b_0tdb_1td

trigrams

$s_0es_1es_2w$, $s_0es_1es_2e$, $s_0es_1eb_0w$, $s_0es_1eb_0td$,
$s_0es_1wb_0w$, $s_0es_1wb_0td$, $s_0ws_1es_2e$, $s_0ws_1eb_0td$

separator

s_0wp, s_0wep, s_0wq, s_0wcq, s_0es_1ep, s_0es_1eq,
s_1wp, s_1wep, s_1wq, s_1weq

extended (Zhu et al., 2013)

s_0llwe, s_0lrwe, s_0luwe, s_0rlwe, s_0rrwe,
s_0ruwe, s_0ulwe, s_0urwe, s_0uuwe, s_1llwe,
s_1lrwe, s_1luwe, s_1rlwe, s_1rrwe, s_1ruwe

disco (Maier, 2015)

s_0xwe, s_1xwe, s_2xwe, s_3xwe,
s_0xtde, s_1xtde, s_2xtde, s_3xtde,
s_0xy, s_1xy, s_2xy, s_3xy
s_0xs_1e, s_0xs_1w, s_0xs_1x, s_0ws_1x, s_0es_1x,
s_0xs_2e, s_0xs_2w, s_0xs_2x, s_0ws_2x, s_0es_2x,
s_0ys_1y, s_0ys_2y, s_0xb_0td, s_0xb_0w

Features from (Tokgöz and Eryiğit, 2015):

counts

s_0P, s_0C, s_0wP, s_0wC, b_0P, b_0C, b_0wP, b_0wC

edges

s_0s_1, s_1s_0, s_0b_0, b_0s_0, s_0b_0e, b_0s_0e

history

a_0, a_1

remote (Novel, UCCA-specific features)

s_0R, s_0wR, b_0R, b_0wR

Figure 1: Binary feature templates for $\text{TUPA}_{\text{Sparse}}$. Notation:
s_i, b_i: ith stack and buffer items.
w, t, d: word form, POS tag and syntactic dependency label of the terminal returned by $h^*(\cdot)$ (see Appendix D).
e: edge label to the node returned by $h(\cdot)$.
l, r (ll, rr): leftmost and rightmost (grand)children.
u (uu): unary (grand)child, when only one exists.
p: unique separator punctuation between s_0 and s_1. q: separator count.
x: gap type ("none", "pass" or "gap") at the sub-graph under the current node.
y: sum of gap lengths (Maier and Lichte, 2009).
P, C: number of parents and children.
R: number of remote children.
a_i: action taken i steps back.

B Extended Presentation of UCCA

This work does not handle two important constructions in the UCCA foundational layer: Linkage, representing discourse relations, and Implicit, representing covert entities. Table 1 shows the statistics of linkage nodes and edges and implicit nodes in the corpora.

	Wiki			20K
	Train	Dev	Test	Leagues
nodes				
# implicit	899	122	77	241
# linkage	2956	263	359	376
edges				
# linkage	9276	803	1094	957

Table 1: Statistics of linkage and implicit nodes in the *Wiki* and *20K Leagues* UCCA corpora. Cf. Table 1.

Linkage. Figure 2 demonstrates a linkage relation, omitted from Figure 1a. The linkage relation is represented by the gray node. LA is *link argument*, and LR is *link relation*. The relation represents the fact that the *linker* "After" links the two parallel scenes that are the arguments of the linkage. Linkage relations are another source of multiple parents for a node, which we do not yet handle in parsing and evaluation.

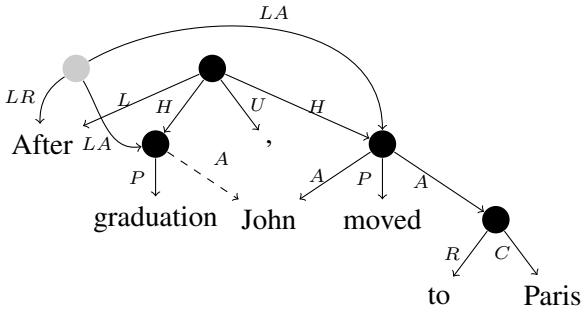

Figure 2: UCCA example with linkage.

Implicit units. UCCA graphs may contain implicit units with no correspondent in the text. Figure 3 shows the annotation for the sentence "A similar technique is almost impossible to apply to other crops, such as cotton, soybeans and rice.". The sentence was used by Oepen et al. (2015) to compare between different semantic dependency schemes. It includes a single scene, whose main relation is "apply", a secondary relation "almost impossible", as well as two complex arguments: "a similar technique" and the coordinated argument "such as cotton, soybeans, and rice." In addition, the scene includes an implicit argument, which represents the agent of the "apply" relation.

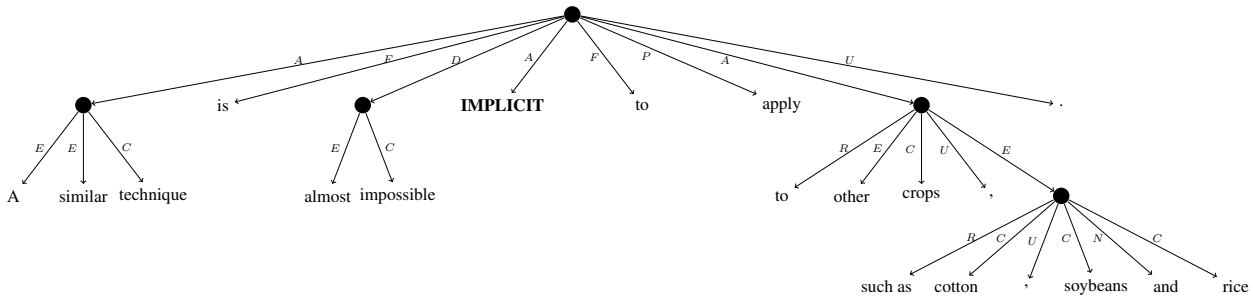

Figure 3: UCCA example with an implicit unit.

The parsing of these units is deferred to future work, as it is likely to require different methods than those explored in this paper (Roth and Frank, 2015).

C Hyperparameter Values

Table 2 lists the hyperparameter values we found for the different classifiers by tuning on the development set. Note that learning rate decay is multiplicative and is applied at each epoch. Mini-batch size is in number of transitions, but a mini-batch must contain only whole sentences.

	Sparse	MLP	BiLSTM
Embedding dimensions			
external word		100	100
word		200	200
POS tag		20	20
syntactic dep.		10	10
edge label		20	20
punctuation		1	1
gap		3	3
action		3	3
Other parameters			
training epochs	19	28	59
MINUPDATE	5		
initial learning rate	1	1	1
learning rate decay	0.1	1	1
MLP #layers		2	2
MLP layer dim.		100	50
LSTM #layers			2
LSTM layer dim.			500
word dropout		0.2	0.2
dropout		0.4	0.4
weight decay		10^{-5}	10^{-5}
mini-batch size		100	100

Table 2: Hyperparameters used for the different classifiers.

D Bilexical Graph Conversion

Here we describe the algorithms used in the conversion referred to in Section 4.

Notation. Let L be the set of possible edge labels. A UCCA graph over a sequence of tokens $w_1, \ldots, w_n$ is a directed acyclic graph $G = (V, E, \ell)$, where $\ell : E \to L$ maps edges to labels. For each token w_i there exists a leaf (*terminal*) $t_i \in V$. A bilexical (dependency) graph over the same text consists of a set A of labeled dependency arcs (t', l, t) between the terminals of G, where t' is the head, t is the dependent and l is the edge label.

Conversion to bilexical graphs. Let $G = (V, E, \ell)$ be a UCCA graph with labels $\ell : E \to L$. The conversion to a bilexical graph requires calculating the set A. All non-terminals in G are removed.

We define a linear order over possible edge labels L (see Figure 4). The priority order generally places core-like categories before adjunct-like ones, and was decided heuristically. For each node $u \in V$, denote by $h(u)$ its child with the highest-priority edge label. The leftmost edge is chosen in case of a tie. Let $h^*(u)$ be the terminal reached by recursively applying $h(\cdot)$ over u. For each terminal t, we define

$$N(t) = \{(u, v) \in E \mid t = h^*(v) \wedge t \neq h^*(u)\}$$

For each edge $(u, v) \in N(t)$, we add $h^*(u)$ as a head of t in A, with the label $\ell(u, v)$. This procedure is given in Algorithm 1.

Data: UCCA graph $G = (V, E, \ell)$
Result: set A of labeled bilexical arcs
$A \leftarrow \emptyset$;
foreach $t \in \mathrm{Terminals}(V)$ **do**
  foreach $(u, v) \in N(t)$ **do**
    $A \leftarrow A \cup \{(h^*(u), \ell(u, v), t)\}$;
  end
end

Algorithm 1: Conversion to bilexical graphs.

Note that this conversion procedure is simpler than the head percolation procedure used for converting syntactic constituency trees to dependency trees (Collins, 1997), since $h(u)$ (similar to u's head-containing child) depends only on $\ell(u, h(u))$ and not on the sub-tree spanned by u, because edge labels in UCCA directly express the role of the child in the parent unit, and are thus sufficient for determining which of u's children contains the head node.

Conversion from bilexical graphs. The inverse conversion introduces non-terminal nodes back into the graph. As the distinction between low- and high-attaching nodes is lost in the conversion, we assume that attachments are always low-attaching. Let A be a the labeled arc set of a bilexical graph. Iterating over the terminals in topological order according to A, we add its members as terminals to graph and create a pre-terminal parent u_t for each terminal t, with an edge labeled as *Terminal* between them. The parents of the pre-terminals are determined by the terminal's parent in the bilexical graph: if t' is a head of t in A, then $u_{t'}$ will be a parent of u_t. We add an intermediate node in between if t has any dependents in A, to allow adding their pre-terminals as children later. Edge labels for the intermediate edges are determined by a rule-based function, denoted by $\mathrm{Label}(t)$. This procedure is given in Algorithm 2.

1. C (Center)
2. N (Connector)
3. H (ParallelScene)
4. P (Process)
5. S (State)
6. A (Participant)
7. D (Adverbial)
8. T (Time)
9. E (Elaborator)
10. R (Relator)
11. F (Function)
12. L (Linker)
13. LR (LinkRelation)
14. LA (LinkArgument)
15. G (Ground)
16. $Terminal$ (Terminal)
17. U (Punctuation)

Figure 4: Priority order of edge labels used by $h(u)$.

E Proof Sketch for Completeness of the TUPA Transition Set

Here we sketch a proof for the fact that the transition set defined in Section 3 is capable of producing any rooted, labeled, anchored DAG. This proves that the transition set is complete with respect to the class of graphs that comprise UCCA.

Let $G = (V, E, \ell)$ be a graph with labels $\ell : E \to L$ over a sequence of tokens $w_1, \ldots, w_n$. Parsing starts with $w_1, \ldots, w_n$ on the buffer, and the root node on the stack.

First we show that every node can be created, by induction on the node height: every terminal (height zero) already exists at the beginning of the parse (and so does the root node). Let $v \in V$ be of height k, and assume all nodes of height less than k can be created. Take any (primary) child u of v: its height must be less than k. If u is a terminal, apply SHIFT until it lies at the head of the buffer. Otherwise, by

Data: list T of terminals, set A of labeled bilexical arcs
Result: UCCA graph $G = (V, E, \ell)$
$V \leftarrow \emptyset$, $E \leftarrow \emptyset$;
foreach $t \in \mathrm{TopologicalSort}(T, A)$ **do**
  $u_t \leftarrow \mathrm{Node}()$;
  $V \leftarrow V \cup \{u_t, t\}$, $E \leftarrow E \cup \{(u_t, t)\}$;
  $\ell(u_t, t) \leftarrow \mathit{Terminal}$;
  foreach $t' \in T, l \in L$ **do**
    if $(t', l, t) \in A$ **then**
      if $\exists t'' \in T, l' \in L : (t, l', t'') \in A$ **then**
        $u \leftarrow \mathrm{Node}()$;
        $V \leftarrow V \cup \{u\}$, $E \leftarrow E \cup \{(u, u_t)\}$;
        $\ell(u, u_t) \leftarrow \texttt{Label}(t)$;
      else
        $u \leftarrow u_t$;
      end
      $E \leftarrow E \cup \{(u_{t'}, u)\}$;
      $\ell(u_{t'}, u) \leftarrow l$;
    end
  end
end
Function `Label`
  Data: node $t \in T$
  Result: label $l \in L$
  if $\mathrm{IsPunctuation}(t)$ **then**
    return *Punctuation*;
  else if $\exists t' \in T : (t, \mathit{ParallelScene}, t') \in A$ **then**
    return *ParallelScene*;
  else if $\exists t' \in T : (t, \mathit{Participant}, t') \in A$ **then**
    return *Process*;
  else
    return *Center*;

Algorithm 2: Conversion from bilexical graphs.

our assumption, u can still be created. Right after u is created, it lies at the head of the buffer. A SHIFT transition followed by a $\text{NODE}_{\ell(v,u)}$ transition will move u to the stack and create v on the buffer, with the correct edge label.

Next, we show that every edge can be created. Let $(v, u) \in E$ be any edge with parent v and child u. Assume v and u have both been created (we already showed that both are created eventually). If either v or u are in the buffer, apply SHIFT until both are in the stack. If both are in the stack but neither is at the stack top, apply SWAP transitions until either moves to the buffer, and then apply SHIFT. Now, assume either v or u is at the stack top. If the other is not the second element on the stack, apply SWAP transitions until it is. Finally, v and u are the top two elements on the stack. If they are in that order, apply $\text{RIGHT-EDGE}_{\ell(v,u)}$ (or $\text{RIGHT-REMOTE}_{\ell(v,u)}$ if the edge between them is remote). Otherwise, apply $\text{LEFT-EDGE}_{\ell(v,u)}$ (or $\text{LEFT-REMOTE}_{\ell(v,u)}$ if the edge between them is remote). This creates (v, u) with the correct edge label.

Once all nodes and edges have been created, we can apply REDUCE until only the root node remains on the stack, and then FINISH. This yields exactly the graph G.

Note that the distinction we made between primary and remote transitions is suitable for UCCA parsing. For general graph parsing without this distinction, the REMOTE transitions can be removed, as well as the single-primary-parent restriction on EDGE transition.

References

Michael Collins. 1997. Three generative lexicalized models for statistical parsing. In *Proc. of ACL*. ACL, Madrid, pages 16–23.

Wolfgang Maier. 2015. Discontinuous incremental shift-reduce parsing. In *Proc. of ACL*. pages 1202–1212. http://aclweb.org/anthology/P15-1116.

Wolfgang Maier and Timm Lichte. 2009. Characterizing discontinuity in constituent treebanks. In *Proc. of Formal Grammar*. Springer, Bordeaux, France, number 5591 in Lecture Notes in Artificial Intelligence, pages 167–182.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinková, Dan Flickinger, Jan Hajič, and Zdeňka Urešová. 2015. SemEval 2015 task 18: Broad-coverage semantic dependency parsing. In *Proc. of SemEval*. pages 915–926. http://aclweb.org/anthology/S15-2153.

Michael Roth and Anette Frank. 2015. Inducing Implicit Arguments from Comparable Texts: A Framework and its Applications. *Computational Linguistics* 41:625–664.

Alper Tokgöz and Gülsen Eryiğit. 2015. Transition-based dependency DAG parsing using dynamic oracles. In *Proc. of ACL Student Research Workshop*. pages 22–27. http://aclweb.org/anthology/P15-3004.

Yue Zhang and Stephen Clark. 2009. Transition-based parsing of the Chinese treebank using a global discriminative model. In *Proc. of IWPT*. Association for Computational Linguistics, pages 162–171. http://aclweb.org/anthology/W09-3825.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proc. of ACL*. pages 434–443. http://aclweb.org/anthology/P13-1043.