Critique on Cache Transition Techniques for Semantic Graph Parsing for optimizing Search Process using Text Mining

Sajini G and Jagadish S Kallimani

ABSTRACT--- This paper elaborates the transition system that gives the standard transition-based dependency parsing techniques for generating the graph. It is essential to know the standard transition techniques for all graphical problems. Cache transition technique plays a vital role in optimizing the search process in various text mining applications. This paper provides an overview on cache transition technique for parsing semantic graphs for several Natural Language Processing (NLP) applications. According to this paper, the cache is having the fixed size m, by tree decomposition theory according to which there is a relationship between the parameter m and class of graphs produced by the theory.

Keywords — Cache Transition Technique, Semantic Graphs, Text Mining, Tree Width, Optimized Search Procedures.

1. INTRODUCTION

The statistical methods were the conventional methods for the natural language efficiency, but are renewed as the algorithms to generate it. These algorithms are almost same as the standard parsing algorithms for getting the syntactic representations. This actually converts the input statement to the graph representations. In recent years, a transition from the chart-based syntactic parsers to stack-based transition systems has increased the efficiency and speed of the real-world applications.

The stack-based transition systems generate graphs than trees by shifting one word at a time onto the stack and then later building all possible arcs between each word on the stack and keeping next word in the buffer. Here in this theory, the graph theoretical notion of tree decomposition has been developed independently in various areas of computer science and discrete mathematics and has been proven useful in discrete optimization and polynomial time algorithms.

1.1 Tree Decomposition and Tree width:

Below is the definition of the Tree Decomposition and Treewidth which has been used along this article, the undirected graph is denoted as G= (V, E) where V is the set of vertices and E is the set of edges. Each edge is represented as an unordered pair (u, v) with u, v ∈ V.

The basic idea for notion of tree decomposition is explained as follows. A tree is a special type of graph was vertices are kept in the hierarchical way with the property that set of vertices in any sub tree have only one interconnection with other vertices. But on the other hand, for general graphs this is not possible and is possible for the graphs where each vertex is interconnected with other vertices. Also, this holds good for the grid-like graph. The notion of tree decomposition of a graph and related notion of tree width gives us the information we need.

The tree decomposition of a graph G is a type of tree having a subset of G's vertices at each node. In a tree decomposition T, the set of nodes is denoted I and the set of arcs is denoted F. The subset of V associated with node i ∈ I is referred to as a bag, and is denoted by Xi. Formally, a tree decomposition of a graph G = (V, E) is defined as a pair ({Xi | i ∈ I}, T = (I, F)) where tree T satisfies all of the following properties.

- Vertex cover: The nodes of the tree T cover all the vertices of G: S i∈I Xi = V. r
- Edge cover: Each edge in G is included in some node of T. That is, for all edges (u, v) ∈ E, there exists an i ∈ I with u, v ∈ Xi . r
- Running intersection: The nodes of T containing a given vertex of G form a connected sub tree. Mathematically, for all i, j, k ∈ I, if j is on the (unique) path from i to k in T, then X_i T $X_k \subseteq X_j$.

The width of a tree decomposition ({Xi}, T) is maxi |Xi | − 1. The treewidth of a graph is the minimum width over all tree decompositions

$$tw(G) = \min_{(\{Xi\},T)\in TD(G)} \max i\,|Xi| - 1$$

where TD(G) is the set of valid tree decompositions of G. they refer to a tree decomposition achieving the minimum possible width as being optimal.

Cache Transition Parser

This is a kind of non-deterministic computational model for graph based parsing. This model takes a sequence of vertices and reads from left to right. This model is based on dependency tree parsing. Here the graph is defined on input vertices, besides its stack and buffer it also uses cache which is a fixed size array of elements. During the computation, vertex in the storage will be either in stack or cache but not in both simultaneously. The graph vertices of input buffer are kept in cache before keeping in stack and these vertices can be directly accessed and edges between them are constructed when in cache.

Revised Manuscript Received on July 10, 2019.

Sajini G, Research Scholar, Department of Computer Science and Engineering, M S Ramaiah Institute of Technology, Bangalore, India. (E-mail: sajini.narayana@gmail.com)

Jagadish S Kallimani, Associate Professor, Department of Computer Science and Engineering, M S Ramaiah Institute of Technology, Bangalore, India. (E-mail: jagadish.k@msrit.edu)

As the cache has fixed-size there should be a fixed number of vertices present when constructing the edges. In order to manage the huge number of vertices and to read the new vertices, the other vertices are kept in buffer and those which are kept in buffer are not accessible to build the edges.

The configuration of the parser can be written as follows,

$$C=(\sigma,\eta,\beta,E)$$

Where,

σ – Stack sequence of vertices and integers with topmost element always at the rightmost position

η – Cache sequence of vertices.

β – Buffer sequence of vertices.

E- the set of edges being built.

The transitions are shown as follows

1. The Push:

This is the parameterized by a position in cache and a set of positions in the cache. This can be represented as (i,C) where i ϵ [m] and C is subset of [m]/{i}

It takes the configuration

$$(\sigma,[v_1,\ldots,v_{i-1},v_i,v_{i+1},\ldots,v_m],v|\beta,E)$$

Then moves to configuration

$$(\sigma|i|v_i,[v_1,\ldots,v_{i-1},v_{i+1},\ldots,v_m,v],\beta,E')$$

$$E'=E\cup\{(v_k,v)\mid k\in C\}$$

2. The Pop :

$$(\sigma|i|v,[v_1,\ldots,v_m],\beta,E)$$

And moves to the configuration

$$(\sigma,[v_1,\ldots,v_{i-1},v,v_i,\ldots,v_{m-1}],\beta,E)$$

Here, the vertex v from the stack is popped up, along with i keeping the origin position in the cache. When we place v in place of i in cache shifting is done.

Lemma 1

Any tree decomposition T of graph G can be transformed into a smooth tree decomposition

T' of G of equal width.

Proof:

Let k be the width of T. At each bag having fewer than k + 1 vertices, continue adding vertices from adjacent bags until all bags have the same size. If two adjacent bags B1 and B2 end up having the same vertices, collapse B1 and B2 into a single bag, and merge the children of the two bags in a way that preserves their order. If two adjacent bags B1 and B2 differ by more than one vertex in their contents, add intermediate bags by adding vertices from B2 and removing vertices from B1 one at a time. Finally, choose a bag B as the root of the tree constructed so far. Add a new root containing k + 1 instances of the special symbol $, and intermediate bags connecting the root to B adding one vertex of B at a time, and removing instances of $.

Now its been introduced with the relative tree width in order of the given order of the vertices of the graph.

Let ,

G=(V,E) be some graph

T some smooth tree decomposition of G

We define the vertex order π(T) of T to be the sequence of vertices produced by visiting T in a preorder. Each vertex of V will appear once in π(T). We need to analyze our parser behavior when a fixed input order is given over the vertices in terms of notion of relative treewidth.

We define the relative treewidth of G against to an order π of G's vertices to keep it minimum width of tree decomposition of G with π order of vertices.

$$\mathbf{rtw}(G,\pi)=\min_{(\{X_i\},T)\in TD(G),\ \pi(T)=\pi}\ \max_i|X_i|-1.$$

Lemma 2

Let c be a configuration of the parser with stack σ and cache η. Let also γ be a minimal reversing sequence of transitions. If we apply to *c* the transitions of γ in the given order, we reach a configuration *c'* with stack σ'= σ and cache η'= η.

Proof:

Let γ =t_1….t_{2S}. If s=1, γ is composed by a push and pop respectively. The definition of pop transition exactly restores the stack and cache of c configuration of push.

If s>1, let γ'=t_2….t_{2S-1}. If γ' is have an equal number of push transitions and pop transitions and not making minimal γ' reversing, then it is split at the point by providing the same reasoning to the two subsequences until it provides minimal reversing.

Assume, c_1 is the configuration due to t_1 applied to c and c_{2S-1} is the configuration due to γ' applied to c_1. On each of the minimal reversing subsequences of γ' we get that the stack and cache of c_1 and c_{2s-1} are equal by using inductive hypothesis. As they are same we can conclude that the pop transition t_{2s} applied to c_{2s-1} and produced with the same configuration of c_{2s}. This will suggest each node of the tree is the configuration of the cache reached at some time step while run. Each push transition is descended from one node to some of its children of the tree, each pop transition return back to its parent, this kind of tree structure is called derivation tree and it represents the history of the parsing process which produces the output graph.

Lemma 3

Consider a cache transition parser with cache size m, and consider a run of the parser with input a vertex sequence _ and with output the constructed graph G. Let T be the derivation tree representing the run. Then T forms a smooth tree decomposition of G having width m-1 and having vertex order π(T)=π.

RESULTS & DISCUSSIONS

Each bag is first created by a push transition, which adds one vertex to the cache and removes one vertex from the cache. Because the bags of T have size m, the size of the cache, the width of T is m -1. Recall that the vertex order π(T) is the sequence of vertices produced by visiting T in a pre-order traversal and listing the vertices newly introduced at the visited bags. Since the derivation tree T is constructed depth first by pushing vertices from the input buffer into the cache, π(T)is exactly the order of the vertices in π.

Lemma 4

Consider a graph G with a smooth tree decomposition T having width m-1, and let π(T) be the vertex order of T. Then T is a derivation tree of a cache transition parser with cache size m, and G is constructed by the associated run given π(T)as input.

Proof.

Let the cache transition parser take a sequence of transitions corresponding to a depth-first traversal of T, pushing an element from π(T) into the cache each time it descends one level in T, and popping each time it ascends. Let (u, v) be an edge of G. Because T is a tree decomposition of G, there is a bag of T containing both u and v. Without loss of generality, let u be the vertex that was introduced before v along the path from the root of T to the bag containing both u and v. Let b_v be the bag at which v is introduced. Because v can only appear in bags in the subtree of T rooted at b_v, this bag containing both u and v must appear in this subtree. Furthermore, by the running intersection property, since u appears in a bag at or below b_v, and is introduced above b_v, u must appear in b_v. Thus, because bags of T correspond to the cache at each step of the parser, the parser's cache will contain u at the step at which v is pushed into the rightmost position of the cache. Therefore, the automaton can build each edge of G. Combining all the 4 lemmas we can formulate the theorems.

Theorem 1

Let G be some graph and let π be some ordering of its vertices. The relative treewidth of G with respect to π is m-1 if and only if a transition parser with input π can construct G using cache size m but not using cache size m-1. The computational problem of deciding whether a transition parser with cache size m and with input π can construct G is treated. Furthermore, the problem of efficiently computing the smallest cache size m that allows a transition parser to construct G from input π is treated.

Theorem 2

A graph G has tree width m-1 if and only if a transition parser with cache size m can construct G for some input ordering of G's vertices, and for no ordering of G's vertices a transition parser with cache size m-1 can construct G.

Oracle Algorithm:

A cache transition computer program is a nondeterministic automaton: For a fastened vertex sequence π, the computer program may construct many graphs, all having tree decompositions with vertex order π (see Lemma 4). Even for a private graph G, there could also be many runs of the computer program on π, every constructing G through a tree decomposition having vertex order π. this is often typically known as spurious ambiguity. In this section we have a tendency to develop associate formula that may be accustomed drive a cache transition computer program with cache size m, in such a approach that the computer program becomes settled. This suggests that at the most one computation is feasible for every combine of G and π. additional exactly, our formula takes as input a configuration c of the computer program obtained once running on π, and a graph G to be created. Then the formula computes the distinctive transition that ought to be applied to c so as to construct G per a canonical tree decomposition of breadth m − one having vertex order π. If such tree decomposition doesn't exist, then the formula fails at some configuration obtained once running on π.

The oracle formulas will cross-check EG so as to choose that transition to use at c, alternatively to choose that it ought to fail. This call relies on 3 reciprocally exclusive rules, listed below. Assume that c has cache η = [v1, . . . , vm] and buffer β. The primary rule is given 1. If there's no edge (vm, v) in EG such vertex v is in β, the oracle chooses transition pop. This rule means that that, as before long as we have a tendency to encounter a vertex in the right position of the cache with no forward-pointing edges (in the input sequence π) that square measure still un- processed, we have a tendency to return to the stack and plan to method different unfinished vertices.

Lemma5

Tree decomposition T of graph G will be remodeled into associate eager tree decomposition T0 of G of equal breadth. Moreover, we've got π(T0) = π(T). Proof: as a result of T is swish tree decomposition, by Lemma four there exists a cache transition computer program with cache size equal to the breadth of T + one.

If the transitions of this run don't violate Rules one to three within the definition of our oracle, then T is additionally associate eager tree decomposition. Just in case the run shows some violations of the 3 rules, we alter T so as to eliminate these violations from the run, during a approach that doesn't increase the width/cache size and preserves the order. Suppose that our run contains some push transition that happens once the right vertex v in the cache η has no forward-pointing edge leading to some vertex in the buffer. This represents a violation of Rule one of the oracle.

Let I be the set of nodes of T, and let i ∈ I be the node of T with right vertex v within the cache, to that this push transition applies; see Figure seven. If there square measure many push transitions out of node i, people who represent a violation of Rule one should all be sorted at the right. We have a tendency to then opt for the right one. Let i1 ∈ I be the node of T created by this push transition, and let T1 be the sub tree of T unmoving at i1. The vertices of G that square measure pushed into the cache within the run related to T1 cannot contain any neighbor of v. therefore v aren't

required in T1. we will thus reattach sub tree T1 to the parent node of i, p(i), in such the simplest way that i1 becomes the immediate right relation of i; see once more Figure seven. what is more, we have a tendency to will replace all occurrences of v in T1 with copies of the vertex introduced at p(i). Let T0 be the tree ensuing from the on top of transformation of T. as a result of our transformation has not modified the dimensions of the luggage of T, T0 continues to be a swish tree decomposition of G, with identical breadth as T. Since our transformation has moved T1 one level up in T while not "jumping over" the other sub tree of T, we have a tendency to should have $\pi(T0) = \pi(T)$.

Note that this transformation of T has far from our run the alleged violation of Rule one. Suppose currently that our run violates Rule two of the oracle. as a result of the run produces G, this will solely happen if the computer program doesn't push into the stack the vertex from the cache that may be required furthest within the future. Let then v1 be the vertex that's pushed onto the stack, and let v2 = v1 be the vertex that is required furthest in the future. Let additionally $i1 \in I$ be the node of T that is created at this step, and let T1 be the sub tree of T unmoving at i1. as a result of v1 is far from the cache once i1 is made, v1 doesn't seem anyplace in T1, and none of the vertices that square measure pushed in T1 square measure neighbors of v1 in G. If v1 isn't a neighbor of the vertices that square measure pushed in T1, then v2 can't be a neighbor of those vertices either, since v2's initial neighbor happens strictly when v1's initial neighbor.

CONCLUSION

Our transition system is impelled by the task of linguistics parsing of linguistic communication sentences, and that we currently proceed to debate a number of the problems that also got to be addressed in developing a sensible system supported our framework. the firsttask is to develop a machine learning system for predicting the parser's next actionat every step. The best cache size can got to be determined through empirical observation, because it could also be beneficial to trade off coverage of the tiny variety of sentences requiring giant cache size so as to create the prediction of programme actions a lot of correct. we have a tendency to speculate that it will be fascinating to decompose the push action into steps that 1st create the choice of whether or not to push or pop, and so whether or not to make every of the potential arcs among the cache severally, so as to cut back the area of predictions at every step. In the literature on dependency synchronic linguistics parsing, models of this kind area unit referred to as arc-factored models and area unit oftimes used. more experimentation are needed to work out the best set of options and the simplest design for the machine learning part.A attainable extension of our framework is that the development of a dynamic programming formula to permit economical exploration of the area of attainable runs of a parser on Associate in Nursing input string. Intuitively, totally different runs on constant string may share common subparts. These subparts is computed just the once, and so "shared" among totally different runs mistreatment dynamic programming techniques. Dynamic programming algorithms for transition-based dependency parsing are planned by Huang and Sagae (2010) and Kuhlmann, Gomez-Rodrıguez, and Satta (2011).

These algorithms may be extended to our system, that is additionally basically stack-based. Dynamic programming algorithms simulating transition-based parsers have tried helpful in the realization of supposed dynamic oracles (Goldberg, Sartorio, and Satta 2014) for transition-based parsers, up parsing performance with relevance static oracles, that is, oracles of the kind mentioned in Section four. what is more, dynamic programming algorithms square measure at the idea of the event of strategies for unattended learning,as for instance the inside-outside formula (Charniak 1993).

Although we've got treated the input buffer as associate degree ordering of the vertices of the final graph, this can be a simplification of the matter setting of linguistics parsing for human language technology.Given as input a sequence of English words, the programme should additionally predict that words correspond to zero, one, or additional vertices of the ultimate graph, and presumably insert vertices not resembling any English word. this might be accomplished either by preprocessing the input string with a separate conception identification part (Flanigan et al.2014), or by extending the actions of the transition system to incorporate moves inserting new vertices into the graph. we've got not enclosed moves inserting new vertices, in order to alter our exposition, however such moves wouldn't basically alter the correspondence between parsing runs and tree decompositions delineate during this article.The correspondence between runs of our programme and tree decompositions of the output graph permits for a particular characterization of the category of graphs coated, as well as straightforward associate degreed economical algorithms for providing an oracle sequence of programme moves, and for determinant the minimum cache size needed to hide an information set. We find through an experiment that linguistics graphs have low relative treewidth with relevanceEnglish order, indicating that our parsing approach provides a sensible methodology of exploiting the order in linguistics parsing. Our conception of relative treewidth with relevance a vertex order seems to be new within the graph theory literature, and may have applications outside of linguistic communication process. Our transition system was primarily intended by these theoretical concerns, and lots of different definitions are attainable. above all, our call that vertices will solely be popped from the rightmost position within the cache simplifies our analysis. Theoretical characterization of, and experimentation with, the set of different attainable transition systems for building graphs may be a promising space for future analysis.

REFERENCES

1. Daniel Gildea, Giorgio Satta, Xiaochang Pengy, University of Rochester Cache Transition Systems for Graph Parsing, Association for Computational Linguistics, 2017, doi:10.1162/COLI a 00308, Volume 44, Number 1.

2. Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse, pages 178–186, Sofia.
3. Choi, Jinho D. and Andrew McCallum. 2013. Transition-based dependency parsing with selectional branching. In Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13), pages 1052–1062, Sofia.
4. Damonte, Marco, Shay B. Cohen, and Giorgio Satta. 2017. An incremental parser for abstract meaning representation. In Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (EACL), pages 536–546, Valencia.
5. Du, Yantao, Fan Zhang, Weiwei Sun, and Xiaojun Wan. 2014. Peking: Profiling syntactic tree parsing techniques for semantic graph parsing. In Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval-2014), pages 459–464, Dublin.
6. Flanigan, Jeffrey, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14), pages 1426–1436, Baltimore, MD.
7. Goldberg, Yoav, Francesco Sartorio, and Giorgio Satta. 2014. A tabular method for dynamic oracles in transition-based parsing. Transactions of the Association for Computational Linguistics, 2:116–130.
8. Flickinger, Dan, Yi Zhang, and Valia Kordoni. 2012. Deepbank: A dynamically annotated treebank of the Wall Street Journal. In Proceedings of the 11th International Workshop on Treebanks and Linguistic Theories, pages 85–96, Lisbon.
9. G´omez-Rodr´ıguez, Carlos and Joakim Nivre. 2013. Divisible transition systems and multiplanar dependency parsing. Computational Linguistics, 39:799–846.
10. Hajic, Jan, Eva Hajicov´a, Jarmila Panevov´a, and Petr Sgall. 2012. Announcing Prague Czech-English dependency treebank 2.0. In Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12), pages 3153–3160, Istanbul.
11. Henderson, James, Paola Merlo, Ivan Titov, and Gabriele Musillo. 2013. Multilingual joint parsing of syntactic and semantic dependencies with a latent variable model. Computational Linguistics, 39(4):949–998.
12. Huang, Liang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL-10), pages 1077–1086, Uppsala.
13. Jones, Bevan, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, and Kevin Knight. 2012. Semantics-based machine translation with hyperedge replacement grammars. In Proceedings of the 24th International Conference on Computational Linguistics (COLING-12), pages 1359–1376, Mumbai.
14. Jones, Bevan K., Sharon Goldwater, and Mark Johnson. 2013. Modeling graph languages with grammars extracted via tree decompositions. In Proceedings of the 11th International Conference on Finite-State Methods and Natural Language Processing (FSMNLP2013), pages 54–62, St. Andrews.
15. Kuhlmann, Marco and Stephan Oepen. 2016. Towards a catalogue of linguistic graph banks. Computational Linguistics, 42(4):819–827.
16. May, Jonathan. 2016. Semeval-2016 task 8: Meaning representation parsing. In Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016), pages 1063–1073, San Diego, CA.
17. Oepen, Stephan, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinkova, Dan Flickinger, Jan Hajic, and Zdenka Uresova. 2015. Semeval 2015 task 18: Broad-coverage semantic dependency parsing. In Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015), pages 915–926, Denver, CO.
18. Peng, Xiaochang, Linfeng Song, and Daniel Gildea. 2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. In Proceedings of the Nineteenth Conference on Computational Natural Language Learning (CoNLL-15), pages 731–739, Beijing.
19. Pitler, Emily and Ryan McDonald. 2015. A linear-time transition system for crossing interval trees. In Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 662–671, Denver, CO.
20. Pourdamghani, Nima, Yang Gao, Ulf Hermjakob, and Kevin Knight. 2014. Aligning English strings with abstract meaning representation graphs. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 425–429, Doha.
21. Ribeyre, Corentin, ´ Eric Villemonte de La Clergerie, and Djam´e Seddah. 2015. Because syntax does matter: Improving predicate-argument structures parsing using syntactic features. In Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-15), pages 64–74, Denver, CO.
22. Wang, Chuan, Nianwen Xue, and Sameer Pradhan. 2015. A transition-based algorithm for AMR parsing. In Proceedings of the 2015 Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL-15), pages 366–375, Denver, CO.
23. Zhou, Junsheng, Feiyu Xu, Hans Uszkoreit, Weiguang Qu, Ran Li, and Yanhui Gu. 2016. AMR parsing with an incremental joint model. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pages 680–689, Austin, TX.