The "Jump and Stay" Method to Discover Proper Verb Centered Constructions in Corpus Lattices

Bálint Sass
Research Institute for Linguistics, Hungarian Academy of Sciences
Budapest, Hungary
sass.balint@nytud.mta.hu

Abstract

The research presented here is based on the theoretical model of corpus lattices. We implemented this as an effective data structure, and developed an algorithm based on this structure to discover essential verbal expressions from corpus data. The idea behind the algorithm is the "jump and stay" principle, which tells us that our target expressions will be found at such places in the lattice where the value of a suitably defined function (whose domain is the vertex set of the corpus lattice) significantly increases (jumps) and then remains the same (stays). We evaluated our method on Hungarian data. Evaluation shows that about 75% of the obtained expressions are correct, actual errors are rare. Thus, this paper is 1. a proof of concept concerning the corpus lattice model, opening the way to investigate this structure further through our implementation; and 2. a proof of concept of the "jump and stay" idea and the algorithm itself, opening the way to apply it further, e.g. for other languages.

1 Introduction

In this paper we present a novel, original verbal construction discovery method. Our starting point will be our former paper (Sass, 2018) which describes a theoretical model considered an appropriate basis for extracting so-called *proper verb centered constructions* from analysed corpora.

First, let us look at our target. In this terminology, verb centered constructions (VCC) are verb + slot structures, where slots can be unfilled (free) or filled (by a filler word). In English, subject (SBJ), direct object (OBJ) and all prepositions can be considered as slots. For example, '*take* + *SBJ* + *OBJ* + *into:account*' has two free slots (SBJ and OBJ) and a filled '*into*' slot where the filler (marked by a colon) is the word '*account*'. In this approach, a filler is the head of the phrase realizes the slot. Length of a VCC (l) is defined as number of slots and fillers added up, the above example has a length of 4.

Then, what are *proper* verb centered constructions (pVCC)? They are complete and clean. That means they contain all necessary elements, and does not contain any unnecessary element for expressing the core meaning of the verbal expression in question. For example, '*take* + *SBJ* + *OBJ:part* + *in*' is proper, while '*take* + *SBJ* + *OBJ:part*' is not proper (because not complete), and '*read* + *SBJ* + *OBJ*' is proper, while '*read* + *SBJ* + *OBJ:book*' is not proper (because not clean). In other words, free slots in pVCCs are complements (subject included) and fillers in pVCCs are idiomatic, carrying some special meaning.

It is clear that pVCCs are constructions. They are form–meaning pairs (Goldberg, 2006; Kay and Michaelis, 2015), they are units of meaning (Teubert, 2005; Danielsson, 2007). Their meaning is assigned to the whole form, they cannot be divided into smaller units if we want to keep the original meaning.

On the other hand, pVCCs are not necessarily multiword. They are multiword in most cases as we have seen in the examples, but there are cases when the multiword property is not satisfied in the strict sense that they consists of two or more whole *words*. Consider for example '*read* + *SBJ* + *OBJ*', it consists of three elements, from which only one is a word, the other two are just slots. Or consider a Hungarian example. In this language slots are specified mostly by bound morphemes, namely case markers. The Hungarian counterpart of '*believe* + *SBJ* + *in*' is '*hisz* + *NOM* + *INE*'

Proceedings of Recent Advances in Natural Language Processing, pages 1076–1084,
Varna, Bulgaria, Sep 2–4, 2019.
https://doi.org/10.26615/978-954-452-056-4_124

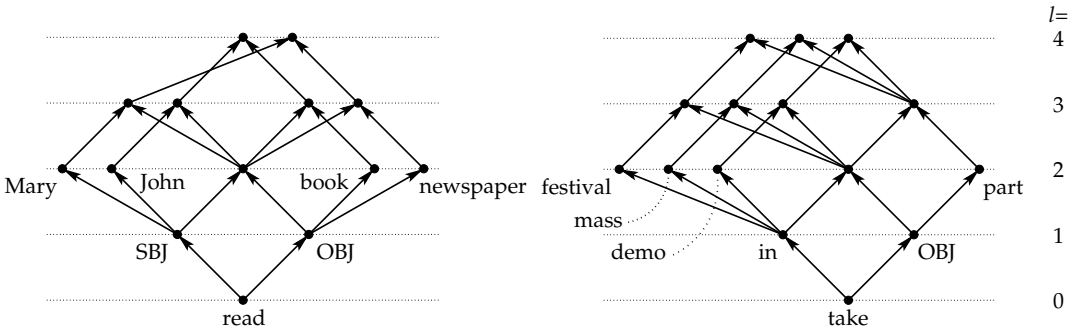

Figure 1: This pair of figures illustrate the notion of double cube and the notion of corpus lattice. On the left, the double cube of '*John reads a book*' ('*read SBJ:John OBJ:book*') and the double cube of '*Mary reads a newspaper*' ('*read SBJ:Mary OBJ:newspaper*') are combined together to create a small corpus lattice. On the right, three clauses are combined. This figure presents two dimensional structures. The subject slot is not depicted in the latter case, it would require three dimensional double cubes.

respectively, where '*NOM*' is for nominative (subject) case, and '*INE*' is for inessive case (appearing as a '*-ban/-ben*' suffix). The English version is strictly multiword, while the Hungarian is not.

We saw that a pVCC should be complete. '*take part*' is a MWE, but '*take + SBJ + OBJ:part + in*' is a pVCC of full value containing all necessary elements. According to Siepmann (2005, page 416) „collocation and verb complementation are intimately related . . . a two-word combination cannot possibly be viewed as a fully-fledged collocation." Simply put, the free slots are just as important as the words/fillers. They turn MWEs into real constructions.

This concept of completeness is essential and unique here. We barely see it neither in classical papers, nor in recent works. Formerly, many papers dealt with just e.g. verb+noun expressions (Evert and Krenn, 2001; Fazly and Stevenson, 2006; Iñurrieta et al., 2016), but recently, also, even the definition of verbal MWEs does not explicitly include the preposition or case marker constituting a complement that, we are convinced, is an inherent part of the expression (Ramisch et al., 2018; Walsh et al., 2018). It is maybe the dependency annotation itself which does not support the approach presented here as case markers are usually not taken as separate units (see Simkó et al., 2017, Fig. 4). Because of the above, we can evaulate our method in itself only.

pVCCs are a large and key group of verbal expressions: they bear the different meanings and usage patterns linked to verbs. We think that it is a good idea if a dictionary presents exactly the set of pVCCs concerning a verb. It is not crucial that they are formally multiword or not. To cover all patterns, we should handle MWEs and constructions uniformly, in one framework. The method presented here shares this attitude.

2 The Initial Model and the Conjecture

Let us summarize the initial model here. The basic processing unit is the clause, the unit which contains a verb together with its complements and adjuncts, and consequently, a pVCC. So as preprocessing, clause boundary detection and some shallow parsing is needed on the input corpus to determine the verb and the top level slots and fillers.

Corpus clauses are represented as so-called *double cubes* (Sass, 2018, Fig. 3), which are a kind of mathematical lattice structures. The verb is at the bottom, every edge adds a slot or a filler to an existing slot (chosen from the clause in question). Vertices are VCCs, they represent nested VCCs of the clause with slots and fillers present or not in all variations. The top represents the original clause: all slots are there and filled as in the original clause. One distinguished vertex is the pVCC.

In the next step, the so-called *corpus lattice (CL)* is created from double cubes containing the same verb. Using a kind of lattice combination operation, double cubes are projected onto each other in a way that where they are identical they will overlap, where they are different they will split up (Fig. 1). It is important that, at the end, the evolving large semilattice structure will represent all clauses of a given verb, and also the distribution of all free and filled slots occurring beside this verb.

The initial paper contains only vague conjectures about how to actually apply the corpus lattice

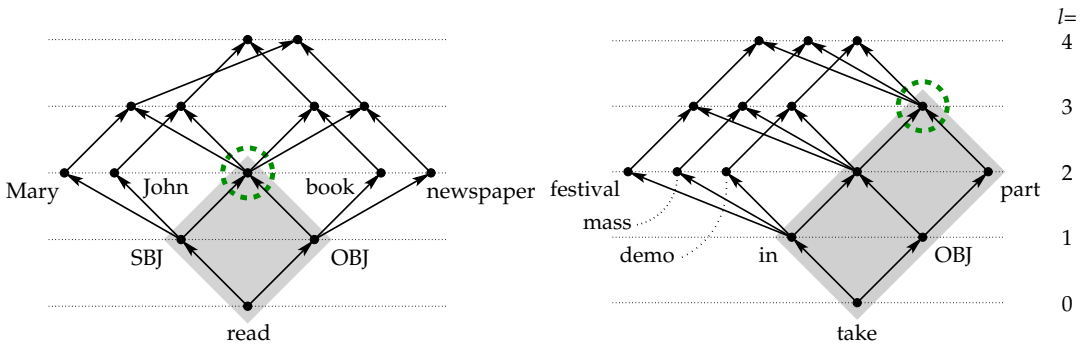

Figure 2: These are the same CLs as in Fig. 1, but here the pVCCs are marked: they are circled. The pVCC is '*read* + *SBJ* + *OBJ*' and '*take* + *SBJ* + *OBJ:part* + *in*' in the two figures respectively. f values are also depicted: $f > 1$ in the gray areas ($f = 2$ in the left lattice and $f = 3$ in the right one), at the other vertices $f = 1$.

structure for discovering pVCCs. We thought, the fact that the corpus lattice contains all information about the distribution of slots and fillers makes it a suitable "representation which can be a basis" for finding pVCCs. We introduced a function (we call it f now) on vertices of the corpus lattice: it is essentially the corpus frequency of the VCC represented by the given vertex. In other words, this f shows how many corpus clauses are represented by the given vertex, or how many corpus clauses this VCC fits to. We formulated the conjecture that pVCCs should be "at some kind of thickening points of the corpus lattice", and added that a future algorithm would move through the corpus lattice somehow systematically to find them.

3 The Idea of "Jump and Stay"

The model described above was purely theoretical. Our current contribution are implementation of the data structure, elaboration and implementation of an algorithm for discovering pVCCs using this data structure, and evaluation of the algorithm on real data.

For outlining our idea which leads to the algorithm, let us take a look at the already known figure from another perspective (Fig. 2). The CLs in the figure are for demonstration purposes: they are, of course, very small, but suitable for presenting the main point (cf. a real CL can be very wide ($\approx$ how many words are there in the corpus), but not too tall ($2 \times$ how many slots are in the longest clause)). We mention that f always grows monotonically downwards in a CL.

Looking at Fig. 2, how can vertices representing pVCCs be characterized? Firstly, as we go top-down in the CL, f suddenly increases at certain points. Secondly, of these vertices, we should prefer those which are located higher in the corpus lattice. As it may be suspected, the first observation will be the basis of "jump" and second one will be the basis of "stay".

The principle of "jump and stay" can be formulated as follows: *jump* means that we advance from a vertex to an adjacent one downwards in the CL if f substantially increases, and *stay* means that we advance from a vertex to an adjacent one upwards in the CL if f remains more or less the same. (Please note, that *stay* also means advancing between vertices, the term itself refers to the fact that the value of f does not change during this step.) In other words, where one (or only few) arrows originate from a vertex, it tend to be a place of stay, similarly, when we have many arrows from the same vertex, it is usually a place of jump. Notice that if we apply the two rules of the principle starting from any of the vertices in Fig. 2, we end up at the circled pVCC. This is the point, this is the main idea of this paper itself.

If we look a bit more closely, in fact, we can end up at one of the top vertices depending on the application order of the two rules, at least in case of the CL on the left. As we will see, top vertices (fully filled clauses) will be excluded from being a pVCC.

pVCCs can be considered as some kind of thickening points indeed, where many edges converge (see the left lattice in Fig. 2), but stating the principle of "jump and stay" is much more clearer.

In addition, we have an independent argument in support of our idea. The "jump and stay" idea

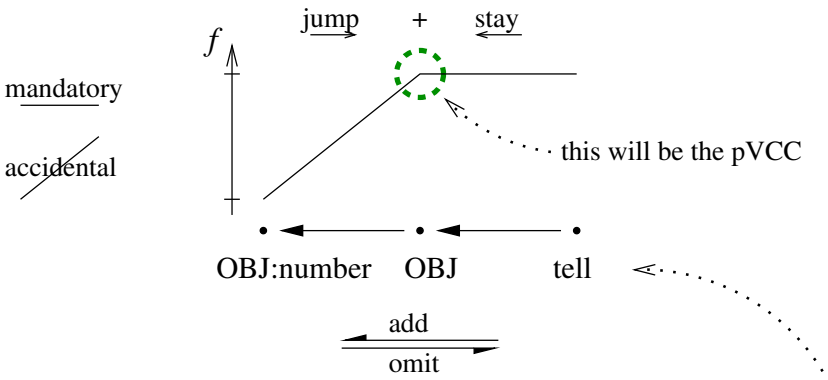

Figure 3: A summing-up of the "jump and stay" principle. A three-vertex piece of the CL of '*tell*' is shown together with the f values for each vertex. It is clear, that this verb requires a direct object, but the direct object itself can be several different words from which '*number*' is quite rare. In this case, there is a stay from '*tell*' to '*tell + OBJ*', and a jump from '*tell + OBJ:number*' to '*tell + OBJ*' so the pVCC is '*tell + OBJ*' here, and that is correct. At the bottom, the directions to add/omit an element to a VCC are shown. On the left, it is shown how a part of the graph of the f function basically look like in the case of mandatory/accidental elements.

is nicely consistent with the fact that constructions have mandatory and accidental elements, some elements are necessary while some are not, as it is also reflected in the definition of the pVCC.

When we jump, we try to omit something which is not mandatory. A typical case of jump, when there are several different fillers (e.g. the various foods as direct object of '*eat*', which are obviously relatively rarer one by one) in a given slot, and the lower vertex is the one, where this slot is free. Advancing to this vertex, we omit this diversity of fillers, we omit something that does not seem to be mandatory (cf. the vertex marked with '*in*' and the three arrows originating from it pointing to the left in Fig. 2).

When we stay, we try to add something which is mandatory. A typical case of stay, when we add an element (a slot or a filler to an existing slot) to a vertex/VCC, but we still cover roughly the same amount of original corpus clauses. This shows, that the added element is mandatory, namely it occurs in nearly all clauses represented by the original VCC. (cf. the vertex marked with '*in*' and the arrows above it in the gray area in Fig. 2).

We always advance by jump downwards (and by stay upwards) in a CL, the other way round – discarding what is needed and adding what is not needed – would not make much sense.

Yet another argument supporting our idea. As we investigate the structure of different CLs, it turns out that a typical pVCC is an endpoint of both jumps and stays, or to put it another way no jump and no stay originate from it (Fig. 2). Plain (not proper) VCCs, however, does not have this property. Almost always, they are a starting point of a jump or a stay (see '*take + OBJ:part + in:festival*' or '*take + OBJ:part*' in Fig. 2 again).

To end this section, look over the "jump and stay" principle in a summary figure (Fig. 3).

Recall the definition of pVCC: stays increase completeness, and jumps increase cleanness. We think that those vertices have the most chance to be a pVCC which can be reached by a stay from below and by a jump from above at the same time.

4 Implementation of the Data Structure

The corpus lattice is a special kind of graph structure. What crucially important to use it effectively is to be able to effectively advance from one vertex to another connected by an edge. For this purpose, we store vertices and edges in hashes (dictionaries) in our python implementation. Edges are stored firstly in one direction, and secondly in the other direction separately.

Starting from a language resource consisting of clauses represented in the form of verb + slots + fillers we build a CL (for each verb separately) as follows:

1. We go through the corpus and take clauses one by one.
2. We build the appropriate double cube of the given clause: starting from the fully filled

clause, we add adjacent edges and vertices omitting slots/fillers one by one recursively.

3. Our graph data structure for a double cube and for a CL both will be the following: we store vertices in a hash, and edges in a hash of hashes (as we have said, in each direction separately). The key of the hashes is a "canonical" JSON string form of the given VCC, its slots ordered alphabetically by slot names.
4. Finally, we combine the current double cube to the corpus lattice being built recording and updating the appropriate f values.

This way we obtain a quite effective representation of a CL.

Input data should be in a specific JSON format which can be generated from either a (shallow) dependency or a (shallow) constituency parse of the input corpus: the verb, the slots and the fillers need to be identified. It is similar to "top level syntactic sequence of the constituent tree" (Shi et al., 2016), with the difference that the order of constituents is not taken into account in our approach.

We used Hungarian data (Sass, 2015) for our experiments. This dataset contains 28 million clauses in a format which was not complicated to convert to the needed input format.

5 The Algorithm

In this section we describe how we implemented the "jump and stay" principle (section 3) using the data structure presented above (section 4).

At the beginning of work, we separated about 7 percent of the data for development purposes. That means, during developing the algorithm we used data only from this part. Developing the algorithm is a kind of learning phase, we draw conclusions based on the input data. It is very important not to use test data for this.

The algorithm consists of the following steps:

1. We go through each vertices of the CL. (The order does not matter, but we chose to begin with the bottom, and continue upwards as pVCCs tend to occur not too far from the bottom.)
2. Some kind of vertices are omitted early on: which are too long (has a length more than 8 ($l > 8$)), which are too rare (has $f < 3$), and which have no out-edge (that means which is at the top of the CL).
3. Firstly, we look for a stay, i.e. try to add a needed element. If the ratio of $f(\text{actual})/f(\text{above}) < 1.7$, then we consider this a stay, and advance to the vertex above. In case of several stays we choose the one with the smallest ratio.
4. Secondly, when no stay can be found, we look for a jump, i.e. try to discard an element which is not needed. If the ratio of $f(\text{below})/f(\text{actual}) > 4$, then we consider this a jump, and advance to the vertex below. In case of several jumps we choose the one with the largest ratio.
5. If we get to a new vertex, we repeat steps 3. and 4.
6. If neither a stay nor a jump can be found, we stop, and if the current VCC is not at the top of the CL (that means it has out-edges) then it is tagged as a pVCC.

Dealing with Hungarian data, at the beginning we do a modification based on Hungarian verb conjugation. Some Hungarian verb suffixes imply that the verb is transitive even when the direct object is not present in the clause. In such cases we add a free OBJ slot. Besides that, Hungarian being a pro-drop language we add a free SBJ slot to every clause without an explicit subject.

A small addition to step 4. In fact, we do not do a jump in every case. If the jump would omit the *last* filler from a VCC, we do not take this step. That is because specific fillers are usually important parts of a pVCC, and full-free VCCs would usually be frequent enough to swallow all pVCCs (being longer only by one filler) performing a jump. So we do the jump only if there remains at least one filler in the resulting VCC or there is no filler in the initial VCC already.

Threshold values (exactly 1.7 for stays, and 4 for jumps) are manually tested and set values. They gave the best results after some experimenting on the development corpus.

Fig. 4 shows some specific examples on how exactly our algorithm works in practice.

The source code of the algorithm and also for building and handling the CL data structure, together with some sample data, is available at `https://github.com/sassbalint/double-cube-jump-and-stay`. The algorithm is fast enough. Building a 365000 vertex CL and investigate it for pVCCs took 63 seconds in total on our server.

```
#4                                                    f= l=
["FAC", null]                                        309  1
 Processing.
 A stay found, we follow.
["FAC", null, "NOM", null]                           309  2
 A stay found, we follow.
["FAC", "jó", "NOM", null]                           307  3
 A stay found, we follow.
["ACC", null, "FAC", "jó", "NOM", null]              300  4
 No stay (ratio=5.17 > 1.7), we stop.
 No appropriate jump (keeping a filler, 1.02 < 4), we stop.
["ACC", null, "FAC", "jó", "NOM", null]              300  4  pVCC

#22699                                                f= l=
["ACC", "költségvetés", "FAC", "jó", "NOM", null]     4  5
 Processing.
 No stay (ratio=2.00 > 1.7), we stop.
 An appropriate jump (keeping a filler, 4<) found, we follow.
["ACC", null, "FAC", "jó", "NOM", null]              300  4
 No stay (ratio=5.17 > 1.7), we stop.
 No appropriate jump (keeping a filler, 1.02 < 4), we stop.
["ACC", null, "FAC", "jó", "NOM", null]              300  4  pVCC
```

Figure 4: Two examples from the output of the algorithm. This demonstrates how the algorithm works: it starts from a vertex and after some jumps and stays it finds the appropriate pVCC in the end. In the first example, we start from a one-free-slot VCC, and get to the pVCC through three stays, adding three mandatory elements (in bold), while the f value decreases from 309 only to 300. In the second example, we start from a longer VCC where also ACC is filled. Here, only one jump is needed, omitting the accidental element '*költségvetés*' ('*budget*') (in bold), to get to the pVCC. (VCCs are in black, additional info is in gray. VCCs are presented here as JSON lists in the form of: slot, filler, slot, filler..., where *null* stands for a free slot.) The verb is '*hagy*' ('*allow*'), input data is taken from the development corpus. As we see, the same pVCC is found in both examples, it is '*hagy + NOM + ACC + FAC:jó*' which is word by word '*allow + SBJ + OBJ + FAC:good*' meaning '*approve + SBJ + OBJ*'. (ACC is for accusative case, FAC is for factive case.) This figure gives a good example of a typical pVCC which "is an endpoint of both jumps and stays", as we said earlier (on page 4).

6 Evaluation and Discussion

The evaluation was carried out in the following manner. Two moderately frequent verbs was chosen: '*húz*' ('*draw/pull*') and '*vet*' ('*cast/throw*'). Their data was taken from the testing part of the corpus (which was 93 percent of the corpus). Our "jump and stay" algorithm was run on these two verbs, and then – according to the f value – the first 20 pVCCs was investigated whether they are correct or not. The input data for these verbs were not only taken from the test corpus, but these verbs were not even looked at in any way during the development phase.

See the results of the evaluation in Table 1. Third column of the table contains the results of the algorithm: the Hungarian pVCCs, fourth column is f value, fifth column is an English translation word by word (or element by element), sixth column is an approximate English counterpart. pVCCs are shown as usual, the verb is taken separately at the top. Slots in Hungarian are marked by the three letter abbreviation of the given case marker: NOM is for nominative (subject) case, ACC is for accusative (direct object) case, and there are some others. Their surface form is not important here, their approximate translation can be seen in the fifth column. Unfilled NOM slots are not shown. (In Hungarian there are also postpositions. Apart from that they are separate words they play similar role as the case markers. Thus,

#	eval	Hungarian pVCC	*f*	word by word	English counterpart
		húz	**9505**	**draw/pull**	
1.	✓	ACC	8304	OBJ	pull sg
2.	✓	ACC:idő	420	OBJ:time	temporize
3.	✓	ACC:haszon + ELA	412	OBJ:profit + from	profit from sg
4.	✓	ACC + SUB:maga	239	OBJ + onto:oneself	put sg on
5.	✓	ACC + után:maga	209	OBJ + after:oneself	pull sg behind oneself
6.	✓	ACC + ALL:maga	207	OBJ + to:oneself	pull/draw sy to oneself
7.	≈	ACC + SUB:fej	199	OBJ + onto:head	put sg on one's head
8.	✓	felé	169	towards	be drawn/attracted towards sg
9.	✓	ACC:rövid	166	OBJ:short	get the worst of it
10.	✓	ACC:vonal	152	OBJ:line	draw a line
11.	✓	ACC:láb	139	OBJ:foot	drag one's feet
12.	✓	ACC:ujj + INS	118	OBJ:finger + with	pick a quarrel with sy
13.	p	ACC + NOM:aki	108	OBJ + SBJ:who	who pulls sg
14.	p	ACC + TEM:az	107	OBJ + at:that	pull sg at that time
15.	✓	ACC + INS:maga	92	OBJ + with:oneself	drag sy/sg with oneself
16.	✓	ACC + felé	85	OBJ + towards	pull sg towards sg
17.	×	ACC + közé	82	OBJ + between	draw sg *(a line)* between sg
18.	✓	ACC:szék	80	OBJ:chair	draw one's chair up
19.	✓	ACC:határ	77	OBJ:border	set limits
20.	✓	ACC:idő + INS	77	OBJ:time + with	temporize on sg
		vet	**14759**	**cast/throw**	
21.	✓	ACC	13649	OBJ	cast/throw sg
22.	≈	ACC + SUB	5437	OBJ + onto	cast/throw sg on sg
23.	✓	ACC:vég + DAT	2632	OBJ:end + for	put an end to sg
24.	✓	ACC + SUB:szem	1085	OBJ + onto:eye	reproach sy for sg
25.	≈	ACC:maga	964	OBJ:oneself	throw oneself
26.	✓	ACC:pillantás + SUB	839	OBJ:glance + onto	glance at sy/sg
27.	✓	ACC + SUB:papír	673	OBJ + onto:paper	note down sg
28.	✓	ACC:fény + SUB	402	OBJ:light + onto	reflect *(well/badly)* on sy/sg
29.	✓	ACC:szám + INS	371	OBJ:number + with	take sg into account
30.	✓	ACC:gát + DAT	362	OBJ:obstacle + for	put a stop to sg
31.	≈	ACC:maga + SUB	345	OBJ:oneself + onto	throw oneself into sg
32.	✓	ACC:maga + ILL	339	OBJ:oneself + into	throw oneself into sg
33.	p	ACC:az + SUB:szem	302	OBJ:that + onto:eye	reproach sy for that
34.	✓	SUB:maga	297	onto:oneself	have only oneself to blame
35.	✓	ACC:szem + SUB	285	OBJ:eye + onto	take a fancy to sy/sg
36.	✓	ACC:kereszt	261	OBJ:cross	cross oneself
37.	✓	ACC:árnyék + SUB	258	OBJ:shadow + onto	cast/throw a shadow over sy/sg
38.	✓	ACC + ILL:lat	240	OBJ + into:*lat*	use sg *(one's power)*
39.	p	ACC + SUB:én	225	OBJ + onto:me	cast/throw sg onto me
40.	p	ACC + NOM:aki	201	OBJ + SBJ:who	who casts/throws sg

Table 1: Evaluation of the "jump and stay" method on '*húz*' ('*draw/pull*') and '*vet*' ('*cast/throw*'). Correct pVCCs are marked with ✓. Further explanation is in the main text.

they have their own slots in some pVCCs, we can find '*után*' ('*after*'), '*felé*' ('*towards*') or '*közé*' ('*between*') in the table.)

Nevertheless, the most important column is the second one which contains the evaluation of the given pVCC in column three. There are four possible values here: ✓ means correct, ≈ means roughly correct, p means contains a pronoun as a filler, and × means not correct (i.e. not complete or not clean).

On the one hand, we see that 70-80 percent of the pVCCs are completely correct, which can be considered a high value in itself. On the other hand, only one single real error is found among 40 constructions which is only 2.5 percent. This one is #17, it is not complete, the direct object slot would be filled by '*vonal*' ('*line*'). The p code indicates a rather trivial problem, which seems to be easily eliminated. Pronouns are very common so they can appear as fillers, but they very rarely bear idiomatic meaning. So the solution could be simply to delete them in a preprocessing step and leave a free slot instead. Note that '*oneself*' and '*each other*' are certainly exceptions here.

Looking through the table, we can make some interesting observations. We see several correct pVCCs, they are complete and also clean. Considering the last column we see that different pVCCs are often translated using completely different verbs. Optionality appears in the form of two (or more) versions of the same construction. #2 and #20 shows essentially the same construction, without and then with a specific complement. This shows that this expression is used both ways, and the ratio of f values (77/420 = 18%) tells us something about which one is how frequent. Constructions #28, #29, and #30 show the importance of our concept of completeness (see secton 1) which takes both collocation and complementation into account. A certain filler often brings in a certain complement, and a new complement is often a sign of a new pVCC.

7 Conclusion and Future Work

Taking the theoretical model of double cubes and corpus lattices we created a new method for discovering useful verbal expressions in corpora. Our idea is called "jump and stay" (see section 3) because, in simple terms, wandering through the corpus lattice the value of a certain function jumps up and then stays the same at certain locations, and these are the locations which points to our target expressions, the so-called proper verb centered constructions. These constructions are proper in the sense that they contain exactly the necessary elements. Consisting of a verb plus slots and fillers, they can be simple or even quite complex; they are not necessarily MWEs, but they are constructions indeed. The evaluation revealed that at least 70-80 percent of the obtained expressions are pVCCs. We worked with Hungarian data, but it would be more or less straightforward to experiment with other languages.

An encouraging feature of the algorithm that it provides complete expressions most of the time (see section 6), incomplete VCCs rarely turn up as pVCCs. However, it has limitations as well. We mentioned the problem with pronouns, another one that there is definitely place to work out some more sophisticated process for setting the threshold values for jumps and stays, but take a look at a more general observation now. In simple cases, if a stay is found (that means an additional element is needed), we add it to the VCC in question doing the step in the corpus lattice defined by this stay (cf. first listing in Fig. 4). But what if we have two (or more) potential additional elements which are not significant separately, but together (their f values added up) they would define a regular stay? In other words, what to do when two (or more) elements seem to be mutually exclusively mandatory at one point? This question can result in some incomplete pVCCs now, and solving this is a promising development direction.

Our conclusion is that the original idea works. The present implementation can be considered as a proof of concept with respect to the corpus lattice model. Clearly, properties of the corpus lattice refer to where pVCCs are located, the introduced lattice structure turned out to be suitable to find them. On the other hand, the "jump and stay" principle also proved to be promising. The basic algorithm presented here can be improved in several aspects, and also the properties, the natural structure of corpus lattices (of given verbs or verb classes) can be further investigated, explored and taken advantage of in the future.

Acknowledgement. This research was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences (case number: BO/00064/17/1; duration: 2017-2020).

References

Pernilla Danielsson. 2007. What constitutes a unit of analysis in language? *Linguistik online* 31(2).

Stefan Evert and Brigitte Krenn. 2001. Methods for the qualitative evaluation of lexical association measures. In *Proceedings of the 39th Meeting of the Association for Computational Linguistics*. Toulouse, France, pages 188–195.

Afsaneh Fazly and Suzanne Stevenson. 2006. Automatically constructing a lexicon of verb phrase idiomatic combinations. In *Proceedings of the 11th Conference of the EACL*. Trento, Italy, pages 337–344.

Adele E. Goldberg. 2006. *Constructions at Work*. Oxford University Press.

Uxoa Iñurrieta, Arantza Diaz de Ilarraza, Gorka Labaka, Kepa Sarasola, Itziar Aduriz, and John Carroll. 2016. Using linguistic data for English and Spanish verb-noun combination identification. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*. The COLING 2016 Organizing Committee, Osaka, Japan, pages 857–867. https://www.aclweb.org/anthology/C16-1082.

Paul Kay and Laura A. Michaelis. 2015. How constructions mean. In *Proceedings of the 11th Workshop on Multiword Expressions*. Association for Computational Linguistics, Denver, Colorado, page 44. https://doi.org/10.3115/v1/W15-0907.

Carlos Ramisch, Silvio Ricardo Cordeiro, Agata Savary, Veronika Vincze, Verginica Barbu Mititelu, Archna Bhatia, Maja Buljan, Marie Candito, Polona Gantar, Voula Giouli, Tunga Güngör, Abdelati Hawwari, Uxoa Iñurrieta, Jolanta Kovalevskaitė, Simon Krek, Timm Lichte, Chaya Liebeskind, Johanna Monti, Carla Parra Escartín, Behrang QasemiZadeh, Renata Ramisch, Nathan Schneider, Ivelina Stoyanova, Ashwini Vaidya, and Abigail Walsh. 2018. Edition 1.1 of the PARSEME shared task on automatic identification of verbal multiword expressions. In *Proceedings of the Joint Workshop on Linguistic Annotation, Multiword Expressions and Constructions (LAW-MWE-CxG-2018)*. Association for Computational Linguistics, Santa Fe, New Mexico, USA, pages 222–240. https://www.aclweb.org/anthology/W18-4925.

Bálint Sass. 2015. 28 millió szintaktikailag elemzett mondat és 500000 igei szerkezet [28 million syntactically annotated sentences and 500000 verbal expressions]. In *XI. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY2015)*. Szeged: JATEPress, pages 303–308.

Bálint Sass. 2018. A lattice based algebraic model for verb centered constructions. In Petr Sojka, Aleš Horák, Ivan Kopeček, and Karel Pala, editors, *Text, Speech and Dialogue*, Springer, Berlin Heidelberg New York, pages 231–238. Lecture Notes in Computer Science, Vol. 11107.

Xing Shi, Inkit Padhi, and Kevin Knight. 2016. Does string-based neural MT learn source syntax? In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, pages 1526–1534. https://doi.org/10.18653/v1/D16-1159.

Dirk Siepmann. 2005. Collocation, colligation and encoding dictionaries. Part I: Lexicological aspects. *International Journal of Lexicography* 18(4):409–444.

Katalin Ilona Simkó, Viktória Kovács, and Veronika Vincze. 2017. USzeged: Identifying verbal multiword expressions with POS tagging and parsing techniques. In *Proceedings of the 13th Workshop on Multiword Expressions (MWE 2017)*. Association for Computational Linguistics, Valencia, Spain, pages 48–53. https://doi.org/10.18653/v1/W17-1705.

Wolfgang Teubert. 2005. My version of corpus linguistics. *International Journal of Corpus Linguistics* 10(1):1–13.

Abigail Walsh, Claire Bonial, Kristina Geeraert, John P. McCrae, Nathan Schneider, and Clarissa Somers. 2018. Constructing an annotated corpus of verbal MWEs for English. In *Proceedings of the Joint Workshop on Linguistic Annotation, Multiword Expressions and Constructions (LAW-MWE-CxG-2018)*. Association for Computational Linguistics, Santa Fe, New Mexico, USA, pages 193–200. https://www.aclweb.org/anthology/W18-4921.