GRAPH QUOTIENTS: A TOPOLOGICAL APPROACH TO GRAPHS

LINAS VEPSTAS

ABSTRACT. This document develops some general concepts useful for extracting knowledge embedded in large graphs and similar cause-effect-type datasets. It attempts to make almost no underlying assumptions, other than that the data can be presented in terms of pair-wise relationships between objects/events. The resulting extracted structures or "patterns" are manifestly symbolic in nature, as they capture and encode the graph structure of the dataset in terms of a (generative) grammar.

INTRO

This document presents some definitions and vocabulary for working with datasets that contain complex relationships, applicable to a large variety of application domains. The concepts borrow from graph theory, and several other areas of mathematics. The goal is to define a way of thinking about complex graphs, and how they can be simplified and condensed into simpler graphs that "concentrate" embedded knowledge into a more manageable size.

The ideas described here are not terribly complex; they represent a kind-of "folk knowledge" generally known to a number of practitioners. However, I am not currently aware of any kind of presentation of this knowledge, either in review/summary form, or as a fully articulated text. There are some texts that discuss these ideas, but they tend to occur primarily in highly abstract mathematical texts, outside of the mainstream computer-science and data-analysis domain. Thus, this document tries to provide an introduction to these concepts in a plain-spoken language. The hope is to be precise enough that there will be few complaints from the mathematically rigorous-minded, yet simple enough that "anyone" can follow through and understand.

Some examples will be provided, primarily drawn from linguistics. However, the concepts are generally applicable, and should prove useful for analyzing any kind of dataset with complex and hidden cause-and-effect relationships. This includes genomic and proteomic data, social-graph data, and even such generic domains as determining the effectiveness of educational curricula or analyzing the narratives of books and movies.

The penultimate can serve as a general example. When teaching students, one never teaches advanced topics until foundations are laid. Yet many students struggle. Given raw data on a large sample of students, and the curricula they were subjected to, can one discern sequences and dependencies of cause-and-effect in this data? Cam one find the most effective curriculum to teach, that advances the greatest number of students? Can one discover different classes of students, some who respond better to one style than another? My beleif is that these questions can not only be answered, but that the framework described here can be used to uncover this structure.

Date: 31 October 2017.

The approach is "highly mathematical", in that most of this document is devoted to defining certain mathematical structures that can be applied to these problems. The mathematics is inspired by and draws upon concepts from algebraic topology, but, for the most part, tries to avoid the difficult abstractions. The devices that are sketched are frameworks (for the most part) and not algorithms. The hope is that you can "pick your own algorithm" – apply your favorite technique and style, neural nets, whatever, as you wish, to obtain results. The goal is to provide a way of talking about, thinking about and presenting data so that the important knowledge contained in it is captured and described, boiled down to a manageable, workable state from a large raw dump of data.

Currently, the ideas described here are employed in a machine-learning project that attempts to extract the structure of natural language in an unsupervised way. Thus, the primary, detailed examples will come from the natural language domain. The theory should be far more general than that.

This document resides in, accompanies source code that implements the ideas here. Specifically, it is in https://github.com/opencog/atomspace/tree/master/opencog/sheaf and it spills over into other files, such as https://github.com/opencog/opencog/blob/master/opencog/nlp/learn/scm/gram-class.scm This code is in active development, and is likely to have changed by a lot since this was written. This document is *not* intended to describe the code; rather, it is meant to describe the general underlying concepts.

For the mathematically inclined, please be aware that the concepts described here touch on the tiniest tips of some very deep mathematical icebergs, specifically in parsing, type theory and category theory. I have no hope of providing the needed background, as these fields are sophisticated and immense. The reader is encouraged to study these on their own, especially as they are applied in computer science and linguistics. There are many good texts on these topics.

The first part provides a definition of a "section" of a graph. A section is a lot like a subgraph, except that it explicitly indicates which edges were cut to form the subgraph. The next part makes use of this concept of "sections" to show how they can be used to talk about and understand pattern-mining, clustering and quotienting. The next part indicates how such clusters can be understood to be "types", in the formal sense of mathematical type theory. Next follows a brief review of the concept of parsing, as it applies to this context. The ability to parse is what motivates the data discovery to begin with: after extracting patterns from a dataset, parsing is how those patterns can be re-assembled. An important part of pattern mining is the ability to distinguish polymorphic behavior. The final part shows how the system as a whole can be understood to be a kind of a sheaf, borrowing a concept from a different branch of mathematics.

SECTIONS

Begin with the standard definition of a graph.

Definition. A GRAPH $G = (V, E)$ is an ordered pair (V, E) of two sets, the first being the set V of vertices, and the second being the set E of edges. An edge $e \in E$ is a pair (v_1, v_2) of vertices, where every v_k *must* be a member of V. That is, edges in E can only connect vertexes in V, and not to something else. $\diamond$

For directed graphs, the vertex ordering in the edge matters. For undirected graphs, it does not. The subsequent will mostly leave this distinction unspecified, and allow either (or both) directed and undirected edges, as the occasion and the need fits. Distinguishing between directed and undirected graphs is not important, at this point. In most of what

follows, it will usually be assumed that there are no edges with $v_1 = v_2$ (loops that connect back to themselves) and that there is at most one edge connecting any given pair of vertexes. These assumptions are being made to simplify the discussion; they are not meant to be a fundamental limitation. It just makes things easier to talk about and less cluttered at the start. The primary application does not require either construct, and it is straight-forward to add extensions to provide these features. Similar remarks apply to graphs with labeled vertexes or edges (such as "colored" edges, vertexes or edges with numerical weights on them, *etc*). Just keep in mind that such additional markup may appear out of thin air, later on.

Besides the above definition, there are other ways of defining and specifying graphs. The one that will be of primary interest here will be one that defines graphs as a collection of sections. These, in turn, are composed of seeds.

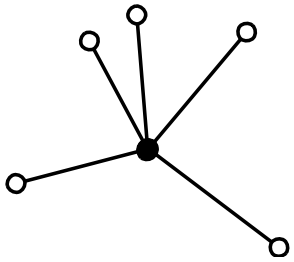
Definition. A SEED is a vertex and the set of edges that connect to it. That is, it is the pair (v, E_v) where v is a single vertex, and E_v is a set of edges containing that vertex, i.e. that set of edges having v as one or the other endpoint. The vertex v may be called the GERM of the seed. $\diamond$

It should be clear that, given a graph G, one can equivalently describe it as a set of seeds (one simply lists all of the vertexes, and all of the edges attached to each vertex). The converse is not "naturally" true. Consider a single seed, consisting of one vertex v_1, and a single edge $e = (v_1, v_2)$. Then the pair (V, E) with $V = \{v_1\}$ and $E = \{(v_1, v_2)\}$ is *not* a graph, because v_2 is missing from the set V. Of course, we could implicitly include v_2 in the collection of vertexes, but this is not "natural", if one is taking the germs of the seeds to define the vertexes of the graph.

Thus, given a seed, each edge in that seed has one "connected" endpoint, and one "unconnected" endpoint. The "connected" endpoint is that endpoint that is v. The other endpoint will commonly be called the CONNECTOR; equivalently, the edge can be taken to be the connector. Perhaps it should be called a half-edge, as one end-point is specified, but missing.

The seed can be visualized as a ball, with a bunch of sticks sticking out of it. A burr one might collect on one's clothing. One can envision a seed as an analog of an open set in topology: the center (the germ) is part of the set, and then there's some more, but the boundary is not part of the set. The vertexes on the unconnected ends of the edges are not a part of the seed.

FIGURE 0.1. A seed



Just as one can cover a topological space with a collection of open sets, so one can also cover a graph with seeds. This analogy is firm: if one has open sets U_i and U_j and $U_i \cap U_j \neq \emptyset$ then one can take U_i and U_j to be vertices, and $U_i \cap U_j$ to be an edge running between them.

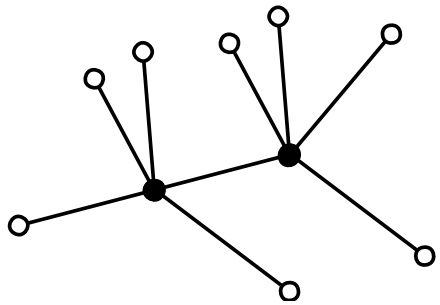
More definitions are needed to advance the ideas of connecting and covering.

Definition. A SECTION is a set of seeds. ⋄

It should be clear that a graph G can be expressed as section; that section has the nice property that all of the germs appear once (and only once) in the set V of G, and that all of the edges in E appear twice, once each in two distinct seeds. This connectivity property motivates the following definition:

Definition. Given a section S, a LINK is any edge (v_1, v_2) where both v_1 and v_2 appear as germs of seeds in S. Two seeds are CONNECTED when there is a link between them. ⋄

FIGURE 0.2. Two linked (connected) seeds



The use of links allows the concepts of paths and connectivity, taken from graph theory, to be imported into the current context. Thus, one can obviously define:

Definition. A CONNECTED SECTION, or a CONTIGUOUS SECTION is a section where every germ is connected to every other germ via a path through the edges. ⋄

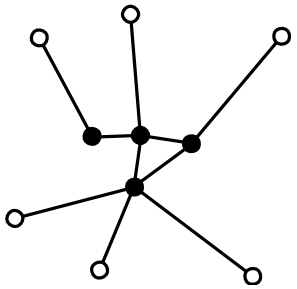
In graph theory, this would normally be called a "connected graph", but we cannot fairly call it that because the seeds and sections were defined in such a way that they are not graphs; they only become graphs when they are fully connected. Never-the-less, it is fairly safe and straight-forward to apply common concepts from graph-theory. Sections are almost like graphs, but not quite.

Note that there are two types of edges in a section: those edges that connect to nothing, and those edges that connect to other seeds in that section. Henceforth, the unconnected edges will be called connectors (as defined above), while the fully-connected edges will be called links (also defined above). Connectors can be thought of as a kind-of half-edge: incomplete, missing the far end, while links are fully connected, whole.

Seeds and sections can (and should!) be visualized as hedgehogs - a body with spines sticking out of it - the connectors can be thought of as the spiny bits sticking out, waiting to make a connection, while the hedgehog body is that collection of vertices and the fully-connected links between them.

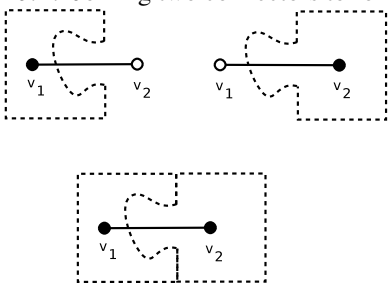
Implicit in the above definitions was that, during link formation, an edge is only allowed to connect to another seed if and only if the connector matches the germ. That is, if (v_1, v_2) is an edge rooted in the seed for v_1 and if (v_3, v_4) is an edge rooted in the seed for v_3, then these two can form a link if and only if $v_2 = v_3$ and $v_4 = v_1$. That is, the connectors are typed: they can only connect to seeds that are of the same type as the unconnected end of the edge.

FIGURE 0.3. A connected section



This motivates a different way of looking at seeds: they can be visualized as jigsaw puzzle pieces, where any given tab on one jigsaw piece can fit into one and only one slot on another jigsaw piece. This union of a tab+slot is the link. The types of the connectors will later be seen to be the same thing as the types of type theory; that is, they are bona-fide types.

FIGURE 0.4. Joining two connectors to form a link



Why sections? Whats the point of introducing this seemingly non-standard approach to something that looks a lot like graph theory? There are several reasons.

- From a computational viewpoint, sections have nice properties that a list of vertexes and edges do not. Given a single seed, one "instantly" know *all* of the edges attached to its germ: they are listed right there. By contrast, given only a graph description, one has to search the entire list E for any edges that might contain the given vertex. Computationally, searching large lists is inefficient, especially so for very large graphs.
- The subset of a section is always a section. This is not the case for a graph: given $G = (V,E)$, some arbitrary subset of V and some arbitrary subset of E do not generally form a graph; one has to apply consistency conditions to get a subgraph.
- A connected section behaves very much like a seed: just as two seeds can be linked together to form a connected section, so also two connected sections can be linked together to form a larger connected section. Both have a body, with spines sticking out. The building blocks (seeds), and the things built from them (sections) have the same properties, lie in the same class. Thus, one has a system that is naturally "scalable", and allows notions of similarity and scale invariance to be explored. There is no need to introduce additional concepts and constructions.

- Given two seeds, one can always either join them (because they connect) or it is impossible to connect them. Either way, one knows immediately. Graphs, in general, cannot be joined, unless one specifies a subgraph in each that matches up. Locating subgraphs in a graph is computationally expensive; verifying subgraph isomorphism is computationally expensive.
- Readers familiar with link-grammar should have by now instantly recognized seeds as being more or less the same thing as "disjuncts" in link-grammar. However, link-grammar disjuncts are a bit more complicated than seeds; this is the topic of the next section.
- In certain ways, the connectors on a seed look like the uncontracted indexes on a tensor. That is, a seed with n connectors on it looks like a tensor of order n. A link looks like a pair of indexes that have been contracted. Tensors have additional properties that seeds do not have; however, the notion of connecting together connectors to form links means that many of the notions from a tensor algebra can be carried over into the current context. This includes, at least partly, the notion that a tensor category describes the algebra of sections.
- The analogy between graphs and topology, specifically, between open sets and seeds, the intersection of open sets and edges, allows concepts and tools to be borrowed from algebraic topology.

If we stop here, not much is accomplished, other than to define a somewhat idiosyncratic view of graph theory. But that is not the case; the concept of seeds and sections are needed to pursue more complex constructions. They provide a tool to study natural language and other systems.

Similar concepts. Linguistics literature sometimes describes similar concepts using a lambda-calculus notation. For example, one can sort-of envision the expression $\lambda M.xyz$ as a seed with the germ M and with connectors x, y and z. This notation has been used to express the concept of a seed, as described above (see Poon & Domingos, for example). The problem with this notation is that, properly speaking, lambda calculus is a system for generating and working with strings, not with graphs, and lambdas are designed to perform substitution (beta-reduction), and not for connecting things.

That is, lambda terms are always strings of symbols, and the variables bound by the lambda are used to perform substitutions. To illustrate the issue, suppose that M above is $axbyczd$ and suppose that $\lambda N.w = ewf$. Can these be "connected" together, linked together like seeds? No: if one tried to "connect" N to z, one has the beta-reduction $(\lambda M.xyz)\lambda N.w \to \lambda axbycewfd.xyw$. There is no way to express some symmetric version of this, because $(\lambda N.w)\lambda M.xyz \to \lambda eaxbyczdf.xyz$ which is hardly the same. Now, of course, lambda calculus has great expressive power, and one could invent a way encoding graph theory, and/or seeds, in lambda calculus; however, doing so would result in verbose and complex system. Its easier to work with graphs directly, and just sleep peacefully with the knowledge that one could encode them with lambdas, if that is what your life depended on.

Note also that there have been extensions of the ideas of lambda calculus to graphs; however, those extensions cling to the fundamental concept of beta reduction. Thus, one works with graphs that have variables in them. Given a variable, one plugs in a graph in the place of that variable. The OpenCog PutLink works in exactly this way. The beta-reduction is fundamentally not symmetrical: putting A into B is not the same as putting B into A. The concept of "connecting" in a symmetric way doesn't arise.

CLUSTERING AND QUOTIENTING

The intended interpretation for the graphs discussed in this document is that they represent or are the result of capturing a large amount of collected raw data. From this data, one wants to extract commonalities and recurring patterns.

The core assumption being made in this section is that, when two local neighborhoods of a graph are similar or identical, then this reflects some important similarity in the raw data. That is, similarity of subgraphs is the be-all and end-all of extracting knowledge from the larger graph, and that the primary goal is to search for, mine, such similar subgraphs.

Exactly what it means to be "similar" is not defined here; this is up to the user. Similarity could mean subgraph isomorphism, or subgraph homomorphism, or something else: some sort of "close-enough" similarity property involving the shape of the graph, the connections made, the colors, directions, labels and weights on the vertexes or edges. The precise details do not matter. However, it is assumed that the user can provide some algorithm for finding such similarities, and that the similarities can be understood as a kind-of "equivalence relation".

If one has an equivalence relation, then the obvious desire, he obvious urge is to attempt to perform quotienting on the graph. That is, to create a new graph, where the "equal" parts are merged into one. The rest of this document describes how this can be done, and what sort of notation to use to do this.

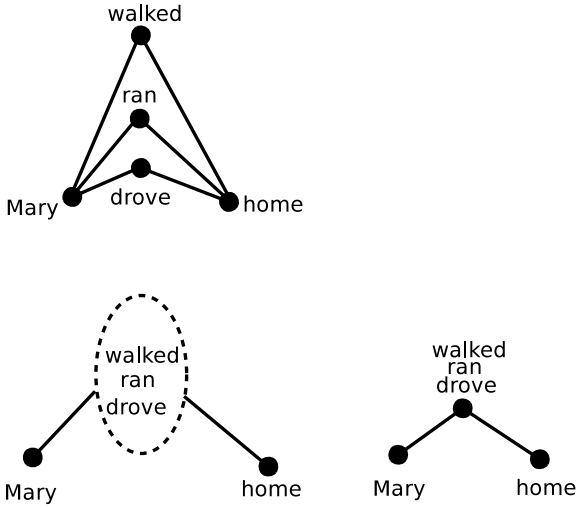
To motivate this, consider the following scenario. One has a large graph, some dense mesh, and one decides, via some external decision process, that two vertexes are similar. One particularly good reason to think that they are similar is that they share a lot of nearest neighbors. In a social graph, one might say they have a lot of friends in common. In genomic or proteomic data, they may interact with the same kinds of genes/proteins. In natural language, they might be words that are synonyms, and thus get used the same way across many different sentences; specifically, the syntactic dependency parse links these words to the same set of heads and dependents. At any rate, one has a large graph, and some sort of equivalence operation that can decide if two vertexes are the "same", or are "similar enough". Whenever one has an equivalence relation, one can apply it to obtain a quotient, of grouping together into an identity all things that are the same.

In graph theory, there is a notion of quotienting, but it is not quite the same as what will be defined shortly. In graph theory, when working with quotients, one will typically consider a graph G relative to some subgraph $A \subset G$. One effectively "draws a dotted line" or places a balloon around the vertexes in A, but preserves all of the edges coming out of A and going into G. The internal structure of A is then typically ignored, discarded; not out of spite, but because that is the nature of an equivalence relation: it states that all elements of A are "equal", are "equivalent", are one and the same, and thus A behaves as if it were a single vertex, with assorted edges attached to it, running from A to the rest of G.

By contrast, the goal here is not just to talk about a graph G relative to a single A, but relative to a huge number of different A's. What's more, the internal structure of these A's will continue to be interesting, and so is carried onwards. Finally, the act of merging together multiple vertexes into one A may result in some of the existing edges being cut, or new edges being created. The clustering operation applied to the graph alters the graph structure. These considerations are what makes it convenient to abandon traditional graph theory, and to replace it by the notion of sheaves and sections.

Given two vertexes v_a and v_b, let s_a and s_b be the corresponding seeds, as defined previously. That is, $s = (v, E_v)$ with E_v being the set of edges connecting v to all of its nearest neighbors. Consider now creating the object $(\{v_a, v_b\}, E_{ab})$. This is no longer a

FIGURE 0.5. Creating a quotient graph



The vertexes "walked", "ran" and "drove" can be considered similar, because they have the same neighbors. The upper graph can be simplified by computing a quotient, shown in the lower diagram: the quotient merges these three similar vertexes into one. The result is not only a simpler graph, but also some vague sense that "walked", "ran" and "drove" are synonymous in some way.

---

seed, as the first item is no longer a single vertex, but a set of vertexes. The set E_{ab} is still a set of edges, depending on the two initial sets of edges E_a and E_b. The precise definition of E_{ab} is not given: it might be the union of E_a and E_b, or the intersection, or some other function. In general, one writes $E_{ab} = f(E_a, E_b)$ for some function f. The result of creating this object is no longer strictly a graph, at least, not in the natural sense (one can force a graph structure onto the result, but doing so gets awkward). The mashing together of two vertexes creates a kind of a quotient, as described above. This quotient will be called a stalk in what follows.

Definition. A STALK is an ordered pair $S = (V, E)$ of vertexes and edges such that every edge in E has one endpoint being a vertex in V and the other endpoint being a vertex not in V. That is, each edge in E is a connector, and no edge in E is a link (back into V). $\diamond$

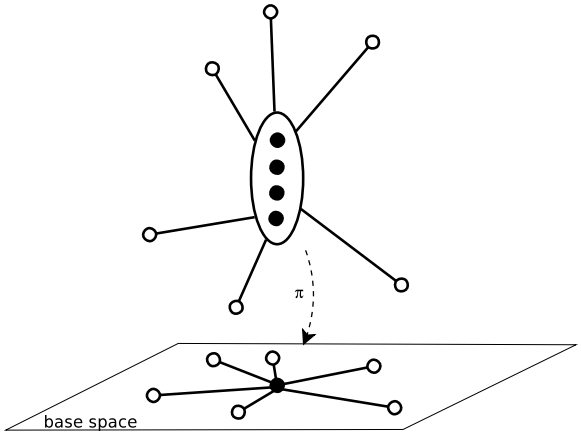
This definition of a stalk is meant to be a straight-forward generalization of the previously defined seed, replacing the germ vertex by a germ that is a set of vertexes. Stalks can be linked together, much as seeds are:

Definition. A LINK between two different stalks $S_a = (V_a, E_a)$ and $S_b = (V_b, E_b)$ is any edge $e = (v_1, v_2)$ running between them, *viz.* where $v_1 \in V_a$ and $v_2 \in V_b$ and $e \in E_a$ and $e \in E_b$. Two stalks are CONNECTED when there are one or more links between them. $\diamond$

It is convenient (it is suggested) that the vertexes in the stalk be visualized as being stacked one on top another, forming a tower or a fiber, with the edges sticking out as spines. Perhaps one can visualize a kind-of melted stack of jigsaw-puzzle pieces. This visualization is suggested only to enforce the idea that two different stalks project down to two different base-points. In particular, one now can have the notion of a meta-graph

where each stalk is a vertex, and each link is an edge. That is, if one flattens the meta-graph down to two dimensions, then one can imagine a stalk growing up as a pole above each meta-vertex, and each meta-edge as being the projection of a link between two stalks. To maintain consistency with standard mathematical terminology, this meta-graph should really be called a "base space", and the stalks and links project down onto it in the usual sense.

FIGURE 0.6. A stalk and it's projection



The projection down to a base space suggests that the equivalence relation on vertexes can be extended to an equivalence relation on edges: two edges are equivalent if they form the same link. That is, one has an equivalence class of edges:

Definition. A LINK between two different stalks $S_a = (V_a, E_a)$ and $S_b = (V_b, E_b)$ is the set $l = \{e_k\}$ of all edges e_k that connect some pair of vertexes in V_a and V_b. That is, every $e_k = (v_{k1}, v_{k2})$ in L has the property that $v_{k1} \in V_a$ and $v_{k2} \in V_b$ and $e_k \in E_a$ and $e_k \in E_b$. $\diamond$

This redefines the notion of a link. Perhaps it should be given a different name, but it should be OK, because the intended sense should be clear from the context. This allows us to redefine the notion of a stalk as well:

Definition. A STALK is an ordered pair $S = (V, L)$ of vertexes and links such that every link in L has one endpoint that is V and the other endpoint not being V. That is, each link in L is a connector or half-edge. $\diamond$

The above definition renders the stalk as being essentially the same thing as a seed, except that now, one is working with sets of vertices, and the links between them.

Why clustering? The above establishes a vocabulary, a means for talking about the clustering of similar things on graphs. It does not suggest how to cluster. Without this vocabulary, it can be very confusing to visualize and talk about what is meant by clustering on a graph. Its worth reviewing some examples.

- In a social graph, a cluster might be a clique of friends. By placing these friends into one group, the stalk allows you to examine how different groups interact with one-another.

- In proteomic or genomic data, if one can group together similar proteins or genes into clusters, one can accomplish a form of dimensional reduction, decreasing the overall size of the dataset. It provides a methodical way of creating a simplified model of biology, without the bad smell of ad-hoc simplifications.
- In linguistic data, the natural clustering is that of words that behave in a similar syntactic fashion; such clusters are commonly called "grammatical classes" or "parts of speech". In particular, it allows one to visualize language as a graph. So: consider, for example, the set of all dependency parses of all sentences in some corpus, say Wikipedia. Each dependency parse is a tree; the vertexes are words, and the edges are the dependencies. Taken as a graph, this is a huge graph, with words connecting to other words, all over the place. Its not terribly interesting in this raw state, because its overwhelmingly large. However, we might notice that all sentences containing the word "dish" resemble all sentences containing the word "plate"; that these two words always get used in a similar or the same way. Grouping these two words together into one reduces the size of the graph by one vertex. Aggressively merging similar words together can sharply shrink the size of the graph to a manageable size. One gets something more: the resulting graph can be understood as encapsulating the structure of the English language.

This last example is worth expanding on. Two things happen when the compressed graph is created. First, that graph encodes the syntactic structure of the language: the links between grammatical classes indicate how words can be arranged into grammatically correct sentences. Second, the amount of compression applied can reveal different kinds of structures. With extremely heavy compression, one might discover only the crudest parts of speech: determiners, adjectives, nouns, transitive and intransitive verbs. Each of these classes are distinct, because they link differently. However, if instead, a lot less compression is applied, then one can discover synonymous words: so, "plate" and "dish" might be grouped together, possibly with "saucer", but not with "cup". Here, one is extracting a semantic grouping, rather than a syntactic grouping.

So, the answer to "why clustering?" is that it allows information to be extracted from a graph, and encoded in a useful, usable fashion. No attempt is made here to suggest how to cluster; merely, that if an equivalence relation is available, and if it is employed wisely, then one can construct quotient graphs that encode important relationships of the original, raw graph.

Similar concepts. One can think of a stalk as a kind of hypergraph, but this view does not seem to be particularly productive.

TYPES

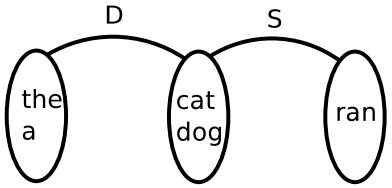
It is notationally awkward to have to write stalks in terms of the sets of vertexes that they are composed of; it is convenient to instead replace each set by a symbol. The symbol will be called a TYPE. As it happens, these types can be seen to be the same things occurring in the study of type theory; the name is justified.

The core idea can be illustrated with link-grammar as an example. The link-grammar disjuncts *are* one and the same thing as stalks. It is worth making this very explicit. A subset of the link-grammar English dictionary looks like this:

```
cat dog: D- & S+;
the a: D+;
ran: S-;
```

This states that "cat" and "dog" are both vertexes, and they are in the same stalk. That stalk has two connectors: D- and S+, which encode the other stalks that can be connected to. So, the D+ can be connected to the D- to form a link. The link has the form ({the, a}, {cat, dog}) and the connector symbols D+ and D- act as abbreviations for the vertex sets that the unconnected end can connect to. The + and - symbols indicate a directionality: to the right or to the left. The capture the notion that, in English, the word-order matters. To properly explain the + and -, we should have to go back to the definition of a graph on the very first page, and introduce the notion of left-right order among the vertices. Doing so from the very beginning would do nothing but clutter up the presentation, so that is not done. The reader is now invited to treat the initial definition of the graph as a monad: there are additional details "under the covers", but they are wrapped up and ignored, and only the relevant bits are exposed. Perhaps the vertices had a color. Perhaps they had a name, or a numerical weight; this is ignored. Here, we unwrap the idea that the vertices must be organized in a left-right order. Its sufficient, for now, to leave it at that.

FIGURE 0.7. Three stalks and two typed links



The three stalks here encode a set of grammatically valid English language sentences. Hooking together the S- and S+ connectors to form an S link, one obtains the sequence [{the, a} {cat, dog} {ran}]. This can be used to generate grammatically valid sentences: pick one word from each set, and one gets a valid sentence. Alternatively, this structure can be taken to encode the sum-total knowledge about this toy language: it is a kind-of graphical representation of the entire language, viewed as a whole.

Definition. Given a stalk $S = (V, L)$, the CONNECTOR TYPE of L is a symbol that can be used as a synonym for the set L. It serves as a short-hand notation for L itself. $\diamond$

Just as in type theory, a type can be viewed a set. Yet, just as in type theory, this is the wrong viewpoint: a type is better understood as expressing a property: it is an intensional, rather than an extensional description. Formally, in the case of finite sets, this may feel like splitting hairs. For an intuitive understanding, however, its useful to think of a type as a property carried by an object, not just the class that the object can be assigned to.

Why types? Types are introduced here primarily as a convenience for working with stalks. They are labels, but they can be useful. Re-examining the examples:

- In a social graph, one group of friends might be called "students" and another group of friends might be called "teachers". The class labels are useful for noting the function and relationship of the different social groups.
- In genomic data, one type of gene sequence might be classified as an exon, another as an intron.

These examples suggest that the use of types is little more than a convenient labeling system. In fact, more hay can be made here, as types interact strongly with category theory: types are used to describe the internal language of monoidal categories. But this is a rather

abstract viewpoint, of no immediate short-term use. Suffice it to say that appearance of types in grammatical analysis of a language is not accidental.

What kind of information do types carry? The above example oversimplifies the notion of types, presenting them as a purely syntactic device. In practice, types also carry semantic information. The amount of semantic information varies inversely to the broadness of the type. In language, coarse-grained types (noun, verb) carry almost no semantic information. Fine-grained types carry much more: a "transitive verb taking a particle and an indirect object" is quite specific: it must be some action that can be performed on some object using some tool in some fashion. An example would be "John sang a song to Mary on his guitar": there is a what, who and and how yoked together in the verb "sang". The more fine-grained the classification, the more semantic content is contained in it.

This suggests that the proper approach is hierarchical: a fine-grained clustering, that captures semantic content, followed by a coarser clustering that erases much of this, leaving behind only "syntactic" content.

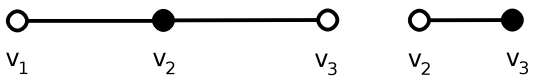
PARSING

The introduction remarked that not every collection of seeds can be assembled in such a way as to create a valid graph. This idea can be firmed up, and defined more carefully. Generically, a valid assembly of seeds is called a parse, and the act of assembling them is called parsing, which is done by parse algorithms. To illustrate the process, consider the following two seeds:

$$v_2 : \{(v_2, v_1), (v_2, v_3)\}$$
$$v_3 : \{(v_3, v_2)\}$$

Represented graphically, these seeds are

FIGURE 0.8. Two unconnected seeds

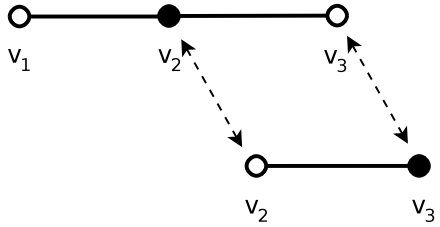


The connector (half-edge) (v_2, v_3) appears with both polarities, and can be linked together to form a link. The connector (v_2, v_1) has nothing to connect to. Even after maximally linking these two seeds, one does not obtain a valid graph: the vertex v_1 is missing from the vertex-set of the graph, even though there is an edge ready to attach to it. This provides an example of a failed parse. It is enough to add the seed $v_1 : \{(v_1, v_2)\}$ to convert this into a successful parse. Adding this seed, and then attempting to maximally link it results in a valid graph; the parse is successful.

Note the minor change in notation: the colon is used as a separator, with the germ appearing on the left, and set of connectors on the right. The relevance of this notational change becomes more apparent, if we label the vertexes in a funny way: let v_1 carry the label "the", and v_2 carry the label "dog" and v_3 carry the label "ran". The failed parse is meant to illustrate that "dog ran" is not a grammatically valid sentence, whereas "the dog ran" is.

Converting these seeds to also enforce left-right word-order requires the notation

FIGURE 0.9. Parsing is the creation of links




```
the: {(the, dog+)}
dog: {(dog, the-), (dog,ran+)}
ran: {(ran, dog-)}
```

This notation is verbose, and slightly confusing. Repeating the germ as the first vertex in every connector is entirely unnecessary. Write instead:

```
the: { dog+ }
dog: { the-, ran+}
ran: { dog- }
```

The set-builder notation is unneeded, and perhaps slightly confusing. In particular, the word "dog" has two connectors on it; both must be connected to obtain a valid parse. The ampersand can be used to indicate the requirement that both connectors are required. This notation will also be useful in the next section.

```
the: dog+ ;
dog: the- & ran+ ;
ran: dog- ;
```

This brings us almost back to the previous section, but not quite. Here, we are working with seeds; previously we worked with stalks. Here, the connector type labels were not employed. In real-world use-cases, using stalks and type labels is much more convenient.

This now brings us to a first draft of a parse algorithm. Given an input set of vertices, it attempts to find a graph that is able to connect all of them.

(1) Provide a dictionary D consisting of a set of unconnected stalks.
(2) Input a set of vertices $V = \{v_1, v_2, \cdots, v_k\}$.
(3) For each vertex in V, locate a stalk which contains that vertex in it's germ.
(4) Attempt to connect all connectors in the selected stalks.
(5) If all connectors can be connected, the parse is successful; else the parse fails.
(6) Print the resulting graph. This graph can be described as a pair (V,E) with V the input set of vertexes, and E the set of links obtained from fully connecting the selected stalks.

The above algorithm is "generic", and does not suggest any optimal strategy for the crucial steps 3 or 4. It also omits discussion of any further constraints that might need to be applied: perhaps the edges need to be directed; perhaps the resulting graph must be a planar graph (no intersecting edges); perhaps the graph must be a minimum spanning tree;

perhaps the input vertexes must be arranged in linear order. These are additional constraints that will typically be required in some specific application.

Why parsing? The benefit of parsing for the analysis of the structure of natural language is well established. Thus, an example of parsing in a non-linguistic domain is useful. Consider having used the above graph compression/vertex-edge clustering techniques to obtain a collection of stalks that describe genomic interactions. This collection provides the initial dictionary D. Now imagine a process where a certain specific set of genes are associated with some particular trait or reaction. Is this a complete set? Can it be said that their interactions are fully understood?

One way to answer these last two questions would be to apply the parse algorithm, using the known dictionary, to see if a complete interaction network can be obtained. If so, then this new specific gene-set fits the general pattern. If not, if a complete parse cannot be found, then one strongly suspects that there remain one or more genes, yet undetermined, that also play a role in the trait. To find these, one might examine the stalks that might have been required to complete the parse: these will give hints as to the specific type of gene, or style of interaction to search for.

Thus, parsing new gene expressions and pathways offers a way of discovering whether they resemble existing, known pathways, or whether they are truly novel. If they seem novel, parsing also gives strong hints as to where to look for any missing pieces or interactions.

Is this really parsing? The above description of parsing is sufficiently different from standard textbook expositions of natural language parsing that some form of an apology needs to be written.

The first step is to observe that the presented algorithm is essentially a simplified, generalized variation of the link-grammar parsing algorithm. The generalization consists in the removal of word-order and link-crossing constraints.

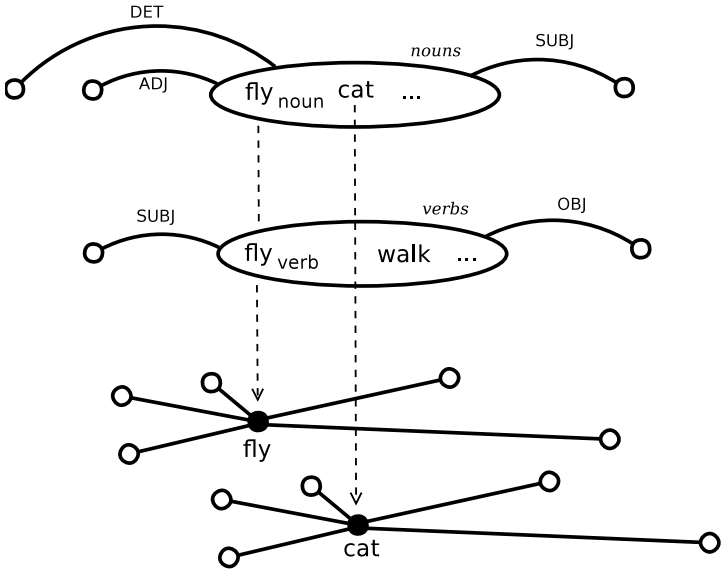
The second step is to observe that the theory of link-grammar is more-or-less isomorphic to the theory of pregroup grammars (see Wikipedia); the primary differences being notational. The left-right directional link-grammar connectors correspond to the left and right adjoints in a pregroup. A link-grammar disjunct (that is, a seed) corresponds to a sequence of types in a pregroup grammar. The correspondence is more-or-less direct, except that link grammar is notational simpler to work with.

The third step is to observe that the link-grammar is a form of dependency grammar. Although the original link-grammar formulation uses undirected links, it is straight-forward and unambiguous to mark up the links with head-dependent directional arrows.

The fourth step is to realize that dependency grammars (DG) and head-phrase-structure grammars (HPSG) are essentially isomorphic. Given one, one can obtain the other in a purely mechanistic way.

The final step is to realize that most introductory textbooks describe parsers for a context-free grammar, and that, for general instructional purposes, such parsers are sufficient to work with HPSG. The priamry issue with HPSG and context-free language parsers is that they obscure the notion of linking together pieces; this is one reason why dependency grammars are often favored: they make clear that it is the linkage between various words that has a primary psychological role in the human understanding of language. It should be noted that many researchers in the psychology of linguistics are particularly drawn to the categorial grammars; these are quite similar to the pregroup grammars, and are more closely related to link-grammar than to the phrase-structure grammars.

FIGURE 0.10. Polymorphism



This figure illustrates a polymorphic assignment for the word "fly". It is split into two parts, the first, a noun, classed with other nouns, showing labeled connectors to determiners, adjectives, and a connector showing that nouns can act as the subject of a verb. The second class shows labeled connectors to subjects and objects, as is appropriate for transitive verbs. Underneath are the flattened raw seeds, showing the words "fly" and "cat" and the myriad of connectors on them. The flattened seeds cannot lead to grammatical linkages, as they mash together into one the connectors for different parts of speech.

POLYMORPHISM

Any given vertex may participate in two or more seeds, independently from one-another. It is this statement that further sharpens the departure from naive graph theory. This is best illustrated by a practical example.

Consider a large graph, constructed from a large corpus of English language sentences. As subgraphs, it might contain the two sentences "there's a fly on your nose" and "did you see it fly?". The vertex "fly" occurs as a noun in one sentence, and a verb in the other. Suppose that the equivalence relation, described in the clustering section, also has the power to discern that this one word should really be split into two, namely fly_{noun} and fly_{verb}, and placed into two different stalks, namely, in the "noun" stalk in the first case, and the "verb" stalk in the second. Recall that these two stalks must be different, because the kinds of connectors that are allowed on a noun must necessarily be quite different from those on a verb. One is then lead to the image shown in figure 0.10.

The point of the figure is to illustrate that, although the "base graph" may not distinguish one variant of a vertex from another, it is important to discover, extract and represent this difference. The concept of "polymorphism" applies, because the base vertex behaves as one of several distinct types in practice. There are several ways the above diagram can be represented textually. As before, the link-grammar-style notation is used, as it is fairly

clear and direct. One representation would be to expose the polymorphism only in the connectors, and not in the base vertex label:

```
fly:  (DET- & ADJ- & SUBJ+) or (SUBJ- & OBJ+);
```

A different possibility is to promptly split the vertex label into two, and ignore the subscript during the parsing stage:

```
fly.noun cat:  (DET- & ADJ- & SUBJ+);
fly.verb walk:  (SUBJ- & OBJ+);
```

Either way, the non-subscripted version of *fly* behaves in a polymorphic fashion.

Note that the use of the notation "or" to disjoin the possibilities denotes a choice function, and not a boolean-or. That is, one can choose either one form, or the other; one cannot choose both. During the parse, both possibilities need to be considered, but only one selected in the end. This implies that at least some fragment of linear logic is at play, and not boolean logic. (this should be expanded upon in future drafts).

SHEAVES

The classification and quotienting procedures cause the structure of the entire network to behave much like the mathematical concept of a "sheaf" studied in "sheaf theory". This is a rather abstract branch of mathematics, most of which offers relatively little insight into the problems being tackled here. Thus, the presentation here is informal, and lacks in rigor. The primary example is again drawn from linguistics, although examples in biochemistry and other fields can be found as well.

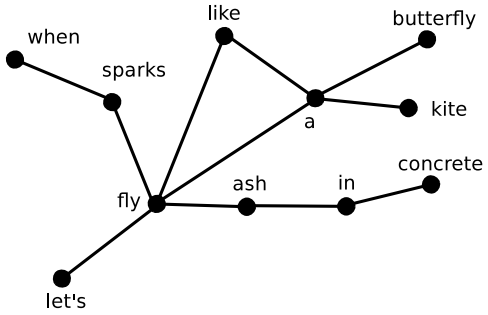
The canonical first step in corpus linguistics is to align text around a shared word or phrase:

```
                  fly like a butterfly
airplanes that    fly
                  fly fishing
                  fly away home
                  fly ash in concrete
   when sparks    fly
         let's    fly a kite
      learn to    fly helicopters
```

Each word is meant to be a vertex; edges are assumed to connect the vertexes together in some way. In standard corpus linguistics, the edges are always taken to join together neighboring words: these then form the "n-grams" that are commonly studied. Alternately, the phrases can be parsed with a dependency parser of one style or another, in which case the words are joined with (directed) edges that denote dependencies. Parsing with a head-phrase parser introduces additional vertexes, typically called NP, VP, S and so on. The resulting graphical structure is shown in the two figures below. The first figure shows alignment based on N-grams, where the phrases are taken as linear sequences of words, without any structure.

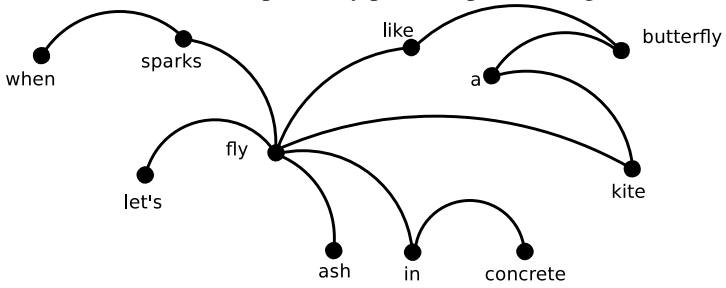
The next figure shows alignment that results from an (unlabeled, undirected) dependency parse of the text. Most noticeable is that the determiner "a" does not link to "fly" even though it stands next to it; instead, the determiner links to the noun it determines. This figure also shows "ash" as modifying "fly", which, as a dependency, is not exactly correct but does serve to illustrate the difference between the N-gram and the dependency alignment.

FIGURE 0.11. N-gram corpus text alignment



N-gram alignment

FIGURE 0.12. Dependency parse corpus text alignment



Dependency alignment

Both of these figures represent a graph that results from a corpus alignment, where all uses of a word have been collapsed (projected down) to a single node. The resulting graph is the graph of the language. One way of visualizing this graph is as a stack of sheets of paper, with one sentence written on each sheet. The papers are stacked in such a way that words that are the same are always vertically above one-another. This "stacking" is where the term "sheaf" comes from. Each single sheet of paper is a "section".

Sections can be cut down, or they can be enlarged, yet each section must consist of a grammatically-correct sentence, phrase (or paragraph, or larger!). The base space consists simply of the individual words in the language, each word being a vertex, connections between words being projections of the connections discovered from the N-grams, or dependency graphs, as the case may be. One can examine how the structure of language changes as words are removed from the base space: such changes correspond to the "restriction morphisms" of a sheaf. These restriction morphisms obey all of the axioms of a sheaf; this observation is what drives the peculiar naming convention given here. The reason this "works", that the axioms apply, is in fact rather shallow: it is because the seeds, as initially defined, behave very much like open sets, and when they are projected to the base space, they serve to cover the base space, much as a topological covering does.

The restriction morphisms appear to continue to satisfy the sheaf axioms even after projection (at least, at the informal level given here).

Why sheaves? The primary reason for introducing this notion is to consolidate the otherwise vague idea of the "language graph". One has dueling notions: the graph of all

sentences; the generative power of grammars. Surface realizations of language are studied in corpus linguistics, where differences in regional dialects, differences according to socio-economic status and politically motivated differences are found. However, these surface realizations are almost never refined into a grammar, and thus, one does not obtain a generative model of how different speakers in different socio-economic classes speak: corpus linguistics examples are just that: examples that are not further refined. By applying a pattern mining approach, the underlying grammar can be discovered computationally. But what is it, really, that is being discovered, other than some collection of grammatical classes and relations? By looking at the collection of all words, sentences and paragraphs in a corpus as if it were a sheaf, one gets a more "holistic" view of what language is: one can start seeing the "big picture", instead of just the trees. This holistic view is the primary point of this exercise.

Related ideas. As before, the "sections" presented above (sentence fragments) are presented minimally. In practice, sections will be adorned with additional information, such as frequency counts and mutual information values. Once clustering and quotienting has been performed, non trivial type tags become available.

CONCLUSION

This document presents a way of thinking about graphs that allows them to be decomposed into constituent parts fairly easily, and then brought together and reassembled in a coherent, syntactically correct fashion. It does so without having to play favorites among competing algorithmic approaches and scoring functions. It makes only one base assumption: that knowledge can be extracted at a symbolic level from pair-wise relationships between events or objects.

It touches briefly, all too briefly, on several closely-related topics, such as the application of category theory and type theory to the analysis of graph structure. These topics could be greatly expanded upon, possibly clarifying much of this content. It is now known to category theorists that there is a close relationship between categories, the internal languages that they encode, and that these are reflections of one another, reflecting through a theory of types. A reasonable but incomplete reference for some of this material is the HoTT book. It exposes types in greater detail, but does not cover the relationship between internal languages, parsing, and the modal logic descriptions of parsing. It is possible that there are texts in proof theory that cover these topics, but I am not aware of any.

This is a bit unfortunate, since I feel that much or most of what is written here is "well known" to computational proof theorists; unfortunately, that literature is not aimed at the data-mining and machine-learning crowd that this document tries to address. Additions, corrections and revisions are welcomed.