Subgraph Queries by Context-free Grammars

Petteri Sevon[1,2] and Lauri Eronen[1]

[1]Helsinki Institute for Information Technology, Department of Computer Science,PO Box 68, FI-00014 University of Helsinki, Finland

Summary

We describe a method for querying vertex- and edge-labeled graphs using context-free grammars to specify the class of interesting paths. We introduce a novel problem, finding the connection subgraph induced by the set of matching paths between given two vertices or two sets of vertices. Such a subgraph provides a concise summary of the relationship between the vertices. We also present novel algorithms for parsing subgraphs directly without enumerating all the individual paths. We evaluate experimentally the presented parsing algorithms on a set of real graphs derived from publicly available biomedical databases and on randomly generated graphs. The results indicate that parsing the connection subgraph directly is much more effective than parsing individual paths separately. Furthermore, we show that using a bidirectional parsing algorithm, in most cases, allows for searching twice as long paths as using a unidirectional search strategy.

1 Introduction

Labeled graphs provide a natural representation for many kinds of structured and heterogeneous data. In such a graph, a path or subgraph (consisting of multiple, usually overlapping paths) between two vertices represents a complex, possibly previously unknown relationship. Link discovery and analysis (for a review on link mining, see, e.g., [1]) aim at finding and evaluating these relationships between entities. Extraction of *connection subgraphs*, interesting or meaningful subgraphs connecting pairs of vertices, is a problem that can be approached from quantitative—find a limited size subgraph that maximizes some metric of reliability, capacity, etc.—or qualitative point of view. In this paper, we focus on the latter; we use *context-free grammars* (CFGs) as a query language to define the class of interesting paths whose union defines the resulting subgraph. The contributions of this paper are (1) formulation of a novel problem—querying connection subgraphs induced by the set of paths matching a CFG—and (2) efficient algorithms for this task.

Publicly available biomedical databases contain vast amounts of rich data, much of which can be interpreted as a labeled graph where vertices correspond to entities and concepts labeled with their types (gene, phenotype, article, etc.) and edges represent known, annotated relationships between vertices labeled with the type of relationship (codes for, interacts with, participates in, etc.). Paths connecting a pair of vertices may correspond to known relationships, but may also reveal novel links, forming the basis for new biological hypotheses.

The subgraph induced by all (length-constrained) paths provides a concise summary of known as well as speculative links between the vertices.

[2]To whom correspondence should be addressed. E-mail: psevon@cs.helsinki.fi

doi:10.2390/biecoll-jib-2008-100

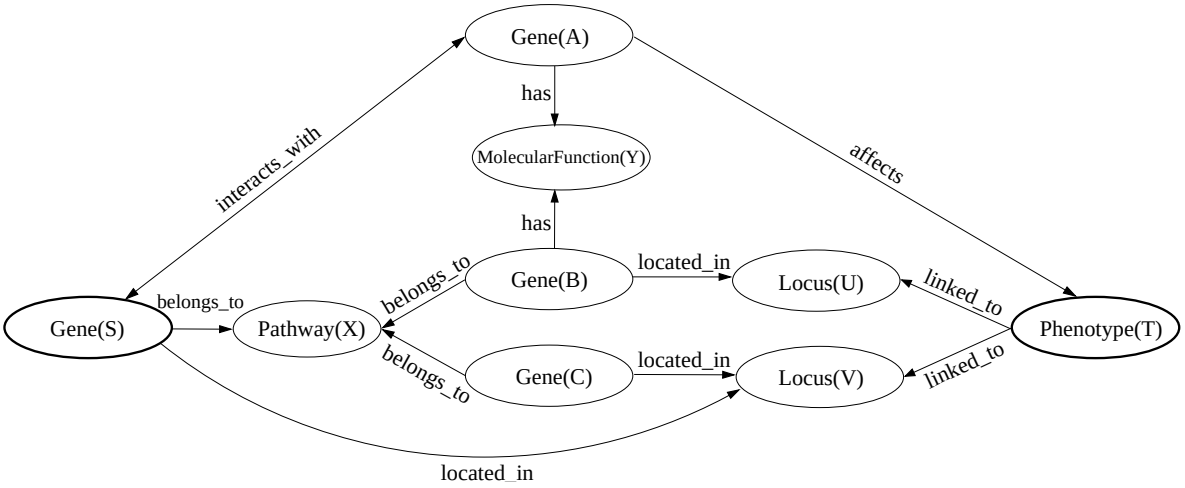


Figure 1: A fictional example connection subgraph summarizing the link between a gene and a phenotype. The user has specified the source and target vertices (Gene(S) and Phenotype(T)), and the class of path types of interest (path types suggesting causal relationship).

Instead of all paths connecting two vertices, an investigator is often interested in paths with specific semantics, e.g, paths that confer similarity or paths that suggest a causal relationship. With labeled graphs, it is natural to base queries on the *path type*—the string of vertex and edge types on a path. A *path class* corresponding to a type of relationship is the set of path types that suggest that type of link between the end-vertices. Figure 1 illustrates the research problem of this paper: extract a small, relevant subgraph from a large graph database induced by paths in a given class. In this example, the subgraph results from a query for causal links from a given gene to a given phenotype.

We propose context-free grammars as a means of defining path classes. A CFG defines a language of strings (path types in our framework) of terminal symbols (edge and vertex types) that can be derived from a distinguished (starting) non-terminal symbol using a set of production rules and a set of other non-terminal symbols. Each non-terminal symbol defines a path class, and paths of a given class are queried by specifying the non-terminal corresponding to that class as the starting symbol. Compared to another common formalism for defining string classes, regular expressions, CFGs are more expressive, and provide a natural means of naming path classes. For the sake of generality, we focus on CFGs in this paper—any regular expression can be easily transformed to an equivalent CFG—but the principle and the presented algorithms can also be applied to regular expressions in a straightforward manner.

Obviously, constructing a dedicated CFG for each query is not always practical. Our proposed subgraph querying system relies on a background CFG, defining a comprehensive domain-specific list of path classes. The complexity of using CFGs is hidden from the end-user: subgraphs can be queried simply by giving either one or two (sets of) end-vertices and the path class of interest. Alternatively, the user can pose more complex queries by manually defining the top level production rules and/or any auxiliary production rules. The proposed system supports two kinds of queries: 1) connection subgraph queries between two given vertices (or sets of vertices), returning the subgraph linking the vertices together, and 2) neighborhood queries, returning the subgraph induced by the set of paths starting from a given vertex and matching the query.

Much of the literature on path queries is based on regular expressions (e.g., [2, 3, 4, 5]). Typically, the goal is to find all objects that can be reached from a given object by following a

path in an edge-labeled graph matching a given regular expression. Mendelson and Wood [6] concluded that finding all vertex-pairs connected by a path matching a regular expression is generally intractable. PQL of Mork et al. [7] is an SQL-like query language, in which one can define complex path classes using rules, which are transformed into a CFG that is subsequently matched against the source knowledge base. Leser's Pathway Query Language [8] is another SQL-like query language for graphs, where the 'WHERE' clause contains path expressions: sequences of vertex variables (that may have other constraints on them) and length-constraints for paths connecting adjacent vertex variables. Context-free grammars can be easily transformed into Prolog programs using a one-to-one mapping from production rules to Prolog clauses. Path querying systems using Prolog have been proposed already in the 1980's, e.g., by Cruz et al. [9].

All of the related work above is focused on evaluating individual paths or listing all paths matching a query; to our knowledge there is no prior work addressing the problem of efficiently retrieving a subgraph induced by a set of paths matching a query specified using a CFG (or other formal languages). A trivial solution to the problem is enumerating all possible paths, parsing each path individually, and returning the union of all matching paths. This approach may in many cases be infeasible due to the number of paths being exponential in the maximum allowed path length in the worst case. We present an improved algorithm that is based on combining many partial parses into a single state, and is guaranteed to work in polynomial time, and show how parsing can be done bidirectionally. Our experiments demonstrate the performance superiority of the improved algorithm, especially using bidirectional search, over the trivial solution.

2 Subgraph queries

Our data model is a directed and labeled graph $G = (V, E)$. The elements of the vertex set V are labeled by a type from a set T_v, such as `Gene` or `Protein`. Edge types from set T_e describe the type of relations between vertices, for example `codes_for` (e.g., gene codes for protein) or `refers_to` (e.g., article refers to gene).

We define the edge set to consist of triplets (u, τ, v), where u and v are vertices from V and $\tau \in T_e$ is the type of the edge between them. Each type τ has inverse $-\tau \in T_e$; for an undirected edge type τ, $-\tau = \tau$. For each edge $e = (u, \tau, v) \in E$ we define its inverse edge $-e = (v, -\tau, u) \in E$ and assume one always exists. The type of path $\mathbf{p} = (v_1, \tau_1, v_2, \tau_2, \ldots, v_k)$ is the sequence of its vertex and edge types: $\text{type}(\mathbf{p}) = (\text{type}(v_1), \tau_1, \text{type}(v_2), \tau_2, \ldots, \text{type}(v_k))$. We also define the *open-ended type* of a path, which is obtained from the path type by removing the vertex types at both ends: $\text{type}_{\text{open}}(\mathbf{p}) = (\tau_1, \text{type}(v_2), \tau_2, \ldots, \tau_{k-1})$.

2.1 Context-free Grammar for Path Types

Formally, a context-free grammar (CFG) is a 4-tuple (Σ, N, P, Q), where Σ and N are sets of terminal and non-terminal symbols, respectively, P is a set of production rules, and Q is the distinguished starting non-terminal. A CFG specifies a class of acceptable strings of alphabet Σ. Production rules specify how strings in the class corresponding to the left-side non-terminal symbol are constructed by concatenating sub-strings. For example, rule `A -> c B d` states

that concatenating terminal c, any string matching non-terminal B, and terminal d produces a string matching non-terminal A. The set of strings accepted by CFG (Σ, N, P, Q) is the set of strings matching Q. The process of checking whether a string is accepted by a CFG is called *parsing*, and a tree formed by applications of production rules to derive the input string from the starting non-terminal is a parse tree. A context-free grammar is unambiguous, if any input string has at most one parse tree.

In our setting, the strings are (possibly open-ended) path-types—alternating sequences of vertex and edge types. We do not have a single string to parse, but a potentially large number of overlapping paths between two given vertices or two sets of vertices.

Terminal and non-terminal symbols map directly onto edge/vertex types and path classes, respectively. The non-terminal symbols are further divided into two groups: ones that have the role of a vertex (vertex-like non-terminals), and one that have the role of an edge (edge-like non-terminals) in a path. The set of vertex-like symbols consists of vertex-like non-terminals and terminal vertex types. The set of edge-like symbols is defined analogously. The right side of a production rule is always an alternating sequence of vertex and edge-like symbols. The right side of a production rule for a vertex-like non-terminal must always begin and end with a vertex-like symbol. The inverse applies to edge-like non-terminals.

We associate non-terminal symbols with path classes: vertex-like or edge-like non-terminal A maps to class $C(\mathtt{A})$ of closed or open-ended path types, respectively. Path $\mathbf{p}$ *matches* vertex-like non-terminal A, if and only if $\text{type}(\mathbf{p}) \in C(\mathtt{A})$, or edge-like non-terminal B, if and only if $\text{type}_{\text{open}}(\mathbf{p}) \in C(\mathtt{B})$.

In this paper, non-terminal symbols are always typed in all capital letters (e.g., IS_SIMILAR_TO). Terminal symbols corresponding to edge types are typed in all small letters (e.g., interacts_with), and terminal symbols corresponding to vertex types are typed with capital initial letters (e.g., Protein).

2.2 Examples

Meaningful path classes that can be defined for edge-like non-terminal symbols include paths conferring similarity, interaction, or causal relationships between pairs of objects. A known interaction between two proteins is reflected in our database as an undirected edge of type interacts_with between the respective vertices. A chain of interacts_with edges is suggestive evidence of a potential pathway of interactions, and we may define an edge-like non-terminal INTERACTS_WITH for such chains with the following production rules:

```
INTERACTS_WITH -> interacts_with |
                  INTERACTS_WITH Protein interacts_with,
```

where the vertical bar is a separator between alternative right-sides of the rule. The first rule states that an edge of type interacts_with matches non-terminal INTERACTS_WITH. The second, recursive rule states that extending a path matching INTERACTS_WITH with an edge of type interacts_with gives a path matching INTERACTS_WITH. By repeated application of the second rule, any length of interaction chains can be obtained. However, the expressive power of context-free grammars is not necessary for this example; the same path class could be defined using a regular expression.

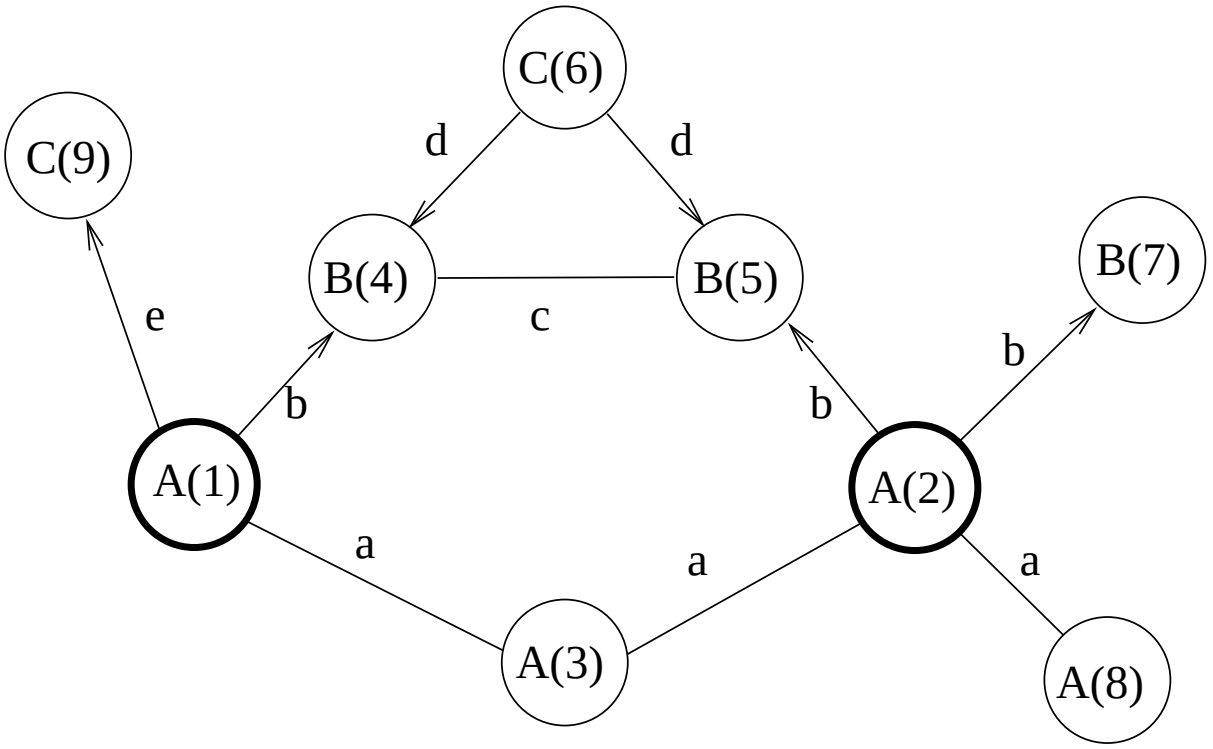


Figure 2: Subgraphs (1,3,4,6) and (2,3,5,6) around the highlighted vertices 1 and 2 (notation `A(1)` means that vertex 1 is of type `A`) are isomorphic and partially overlapping. This results in palindromic paths of types (`A,b,B,-d,C,d,B,-b,A`), (`A,b,B,c,B,-b`) and (`A,a,A,a,A`) connecting the two vertices. Note that the mirror of an asymmetric edge type is its inverse type; in this example edge types `a` and `c` are symmetric, and all other edge types are asymmetric.

The second example illustrates a situation where regular expressions do not have sufficient expressive power. One possibility to define similarity of vertices in a graph is by the similarity of their contexts: two vertices are similar, if they are of the same type and have edges of the same type to similar vertices. The definition is recursive; the similarity of the neighboring vertices depends on the similarity of their neighbors, and so on. It can be shown that this kind of similarity implies isomorphic neighborhoods between the pair of vertices that are partially overlapping, or connected by edges conferring similarity (Figure 2). As a result, the pair of vertices is connected by palindromic paths, which can be recognized with a set of production rules of form `IS_SIMILAR_TO -> e V -e | h` for all meaningful (edge-like symbol, vertex-like non-terminal)-pairs (`e, V`), and for all similarity-conferring edge types `h` (e.g., `is_homologous_to`). We assume that each vertex-like non-terminal `V` corresponding to terminal vertex type `v` has productions `V -> V IS_SIMILAR_TO v | v`, allowing for recursive nesting of the palindrome rule.

Vertex-like non-terminal symbols are useful for several reasons. First, any vertex can be replaced with two vertices connected by a path conferring similarity by using the following construct: `GENE -> GENE IS_SIMILAR_TO Gene | Gene`. These production rules state that terminal vertex type `Gene`, or an arbitrarily long sequence `Gene IS_SIMILAR_TO Gene ...` `Gene` can be derived from the vertex-like non-terminal `GENE`. This kind of rule is specified for all vertex-like non-terminals, and productions are typically given in terms of the non-terminals rather than terminal vertex types. This kind of behavior is desired, since the user is most likely interested in non-trivial, speculative links between the vertices of interest, which can often be obtained by transfer of a property of an object to a similar object.

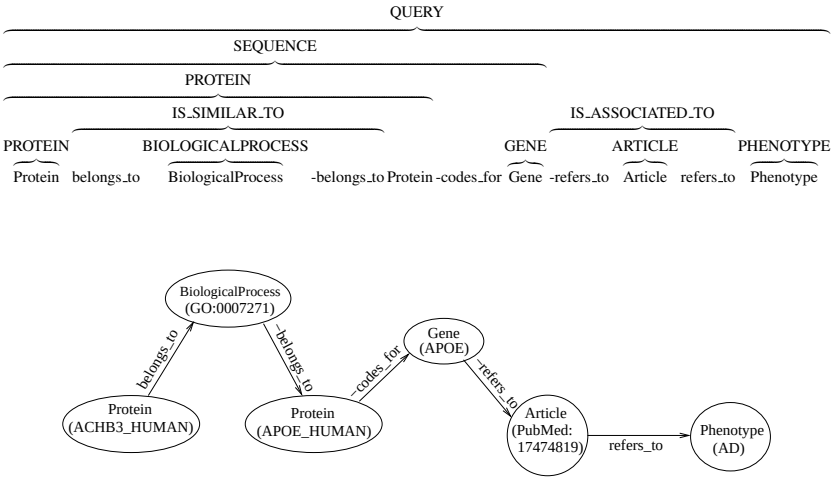


Figure 3: An example illustrating a parse tree for a path type matching a path in the graph database.

In another scenario, the user does not want to make a distinction between, say, a gene and a protein. To implement this, we introduce a new non-terminal `SEQUENCE` encompassing both genes and proteins: `SEQUENCE -> GENE | PROTEIN | GENE codes_for PROTEIN | PROTEIN -codes_for GENE`. In this example, not only can we handle genes and proteins in a uniform manner, but we also allow for paths with an incoming edge to a gene and the outgoing edge from the protein it codes for, or vice versa. This also serves as an example on how taxonomies can be imposed on vertex-types; `SEQUENCE` subsumes both `GENE` and `PROTEIN`. Taxonomies can be formed for edge-like non-terminals and terminals in the same way.

2.3 Querying

We propose a simple syntax for subgraph queries taking as input one or two sets of vertices and possibly an upper limit for path length. Together these specify the set of paths to be considered. In the query, the user also specifies the root level production rules for the starting non-terminal Q = `QUERY` (other non-terminals are defined in a background knowledge file). The following sample query finds paths from protein ACHB3_HUMAN (start-vertices are given in 'FROM' clause) to the Alzheimer disease (end-vertices are given in 'TO' clause) that match non-terminal `IS_ASSOCIATED_TO` and are of length 5 or less edges.

```
FROM Protein(ACHB3_HUMAN)
TO Phenotype(AD)
MAXLEN 5
QUERY -> SEQUENCE IS_ASSOCIATED_TO PHENOTYPE
```

The system returns the connection subgraph induced by all the accepted paths, i.e., contains the edges and only the edges that occur in at least one accepted path. Figure 3 shows the parse-tree corresponding to a matching path. In this example we assume that productions for non-terminals `SEQUENCE`, `IS_SIMILAR_TO`, `GENE`, `PROTEIN`, and `BIOLOGICALPROCESS` are defined as outlined above, and nonterminal `IS_ASSOCIATED_TO` has at least production rule `IS_ASSOCIATED_TO -> -refers_to ARTICLE refers_to` defined for it.

3 Algorithms for Parsing Subgraphs

Our parsing algorithm is based on the well-known Earley parser [10] (the original algorithm handles character strings as input, instead of graphs). The Earley parser can handle arbitrary CFGs, and has time complexity $O(n^3)$ ($O(n^2)$ for unambiguous grammars), where n is the length of the parsed string. Let us first recall the basics of the standard Earley parser (without look-ahead). Throughout this section we denote strings of nonterminal and terminal symbols (including the empty string) by Greek letters, single terminals by small letters and single nonterminals by capital letters. For each position in the input string, the parser maintains a set of states of form $(A \rightarrow \alpha \cdot \beta, n)$, where n is the *origin pointer*, input position at which parsing of the dotted rule began. The interpretation of the *dot condition* $A \rightarrow \alpha \cdot \beta$ is that a parse exists for the substring from position n to the current input position for rule $A \rightarrow \alpha$. If β is the empty string, then the state is *completed*.

The algorithm processes the input string from left-to-right. For each input position i, it iterates over the respective stateset $S(i)$, and performs at most one of three operations for each state $(A \rightarrow \alpha \cdot X\beta, n) \in S(i)$, where X is any symbol or the empty string if the state is completed:

- **Prediction** If X is non-terminal, then add state $(X \rightarrow \cdot\delta, i)$ to $S(i)$ for each rule $X \rightarrow \delta$ in the grammar.
- **Scanning** If X is terminal and the ith input symbol is also X, then add state $(A \rightarrow \alpha X \cdot \beta, n)$ to $S(i+1)$.
- **Completion** If $X\beta$ is the empty string, then the state is completed; add state $(B \rightarrow \delta A \cdot \gamma, m)$ to $S(i)$ for each state $(B \rightarrow \delta \cdot A\gamma, m) \in S(n)$.

At the beginning, stateset $S(1)$ contains *initial states* $(Q \rightarrow \cdot\alpha, 1)$, where Q is the starting non-terminal, for each rule $Q \rightarrow \alpha$ in the grammar. A successful parse is found, when an *accepted state* of form $(Q \rightarrow \alpha\cdot, 1)$ is added to $S(\ell + 1)$, where ℓ is the length of the input string.

3.1 Earley parser for subgraph queries

In this paper, we assume that the complete data graph fits into RAM (stored as adjacency lists), which allows for efficient vertex neighbor queries. We thus concentrate on the parsing aspect of the queries, and not on the physical representation of data. The search proceeds in breadth-first manner in the graph, which means that presented algorithms could also straightforwardly be used for graphs stored into e.g. a relational database, by performing neighborhood queries for all vertices within the same path length in one batch.

To adapt the Earley parser for handling data in graph format, let us first transform the original edge and vertex-labeled graph $G = (V, E)$ into a bipartite edge-labeled graph $G' = (V \cup V', E')$, in which edges of the original graph are represented as edges from V to V', and vertex types are are represented as edges from V' to V. The rationale behind this is that in the transformed graph the parsed strings are strings of edge types only, which eliminates the need to process vertex types and edge types as separate cases.

Formally, let $b : V \rightarrow V'$ be a bijection. Then, $(u, t, v) \in E \Leftrightarrow (u, t, b(v)) \in E'$ and for each $v \in V : (b(v), \text{type}(v), v) \in E'$. All paths in G' alternate between edges that correspond to edges

in G, and edges that correspond to vertices in G. Consequently, a string of edge types of a $V \to V'$ path or a $V' \to V$ path in G' coincides with the open-ended path type or the path type of the corresponding path in G, respectively.

The most straightforward way of applying the Earley parser to graphs is to parse each (length-limited) path between the start-vertices (V_S) and the end-vertices (V_T) as a separate string. A trivial improvement is achieved by organizing paths originating from a starting vertex into a tree, and parsing each path prefix only once. In this kind of algorithm state sets are maintained for all path prefixes (i.e., path prefixes have the role of input position of the standard algorithm). This may still be too expensive computationally, since the number of prefixes can be exponential in the number of vertices visited during parsing.

3.2 Improved parser

The key idea in the improved algorithm presented next is that in any dot condition $A \to \alpha \cdot \beta$ the actual paths matching α traversed from the vertex in which the state was predicted to the current vertex do not affect subsequent parsing (bar cycle elimination). This means that we can avoid redundant work by collapsing partial parses for paths between the same pair of vertices in the same dot condition into a single state. We need to, however, make sure that the origin pointers do not form cycles (except in case of left-recursion) to guarantee correct behavior of the algorithm. This is done by maintaining state set $S(j, v)$ for all (path length, vertex)-pairs: the interpretation of state $(A \to \alpha \cdot \beta, i, u)$ in a state set for a (path length, vertex)-pair (j, v) is that parsing of the dot condition started from vertex u at path length i, and there is at least one path of length $j - i$ from vertex u to v that satisfies rule $A \to \alpha$.

A drawback of this algorithm is that cycles can only be avoided at the level of a single non-terminal; e.g., in rule $A \to aBa$ for an edge-like nonterminal A, we can guarantee that the first and last vertex on the path matching A are distinct vertices, but we cannot effectively test if there is a path matching a non-terminal B that does not visit either end of the path matching A.

At the beginning, state sets $S(0, v)$ for all start-vertices $v \in V_S$ are initialized to contain states $(Q \to \cdot\alpha, 0, v)$, where Q is the starting non-terminal, for each rule $Q \to \alpha$ in the grammar. When state $s = (Q \to \alpha\cdot, 0, v)$, where $v \in V_S$, is added to $S(\ell, u)$, where $u \in V_T$, state s is *accepted*, i.e., corresponds to a successful parse. For neighborhood queries, condition $u \in V_T$ is ignored (V_T is not specified).

Completion and prediction are no different from the standard Earley parser, except that completion is not performed for state $(A \to \alpha\cdot, i, u) \in S(j, v)$, if (1) A is an edge-like non-terminal and $v = b(u)$, or (2) A is a vertex-like non-terminal, $u = b(v)$ and $j - i > 1$. The purpose of these conditions is to eliminate cycles where a path matching a non-terminal starts and ends at the same vertex in graph G. Scanning is performed as follows: If t is terminal in state $(A \to \alpha \cdot t\beta, i, u) \in S(j, v)$, then add state $(A \to \alpha t \cdot \beta, i, u)$ to all sets $S(j + 1, w)$ such that $(v, t, w) \in E'$.

The algorithm needs to proceed in breadth-first order to work correctly: all scanning operations that add states to any given state set must be performed before processing the state set. First, all state sets at path length 0 are processed, then all state sets at path length 1, and so on.

The subgraph induced by all accepted paths is the set of edges scanned in any parse leading to an accepted state. Obtaining this set requires backtracking the parse graph induced by the origin

pointers. First, we add each accepted state $s \in S(i,u)$ to the corresponding set of accepted states $A(i,u)$. Next we process the accepted state sets in reverse order of path length using inverse operations of scanning and completion. For each state $(A \rightarrow \alpha X \cdot \beta, i, u) \in A(j,v)$:

- **Reverse scanning** If X is terminal: For each state $s = (A \rightarrow \alpha \cdot X\beta, i, u) \in S(j-1, w)$ such that $(w, X, v) \in E'$ add s to $A(j-1, w)$, and add edge (w, X, v) to the resulting connection subgraph.
- **Inverse prediction** If X is non-terminal: For each completed state $s_1 = (X \rightarrow \delta \cdot, k, w) \in S(j,v)$, if state $s_2 = (A \rightarrow \alpha \cdot X\beta, i, u) \in S(k,w)$, then add s_1 to $A(j,v)$ and s_2 to $A(k,w)$.

Worst case time complexity of the algorithm is $O(n^3m^3)$, where $n = |V \cup V'|$ and m is the maximum path length: there are at most nm state sets, each containing at most $O(nm)$ states (the number of possible dot conditions is a constant), out of which at most $O(nm)$ are completed, and each completed state may have $O(nm)$ states from which it is predicted. The backtracking phase performs inverse operations only for a subset of the operations of the forward phase, and thus does not affect the asymptotic time complexity. In practice, typically only a small portion of all possible states is explored by the algorithm, and the results in Section 4 give a better picture of the real-world performance of the algorithm than the cubic asymptotic worst-case time complexity.

3.3 Bidirectional parsing

Our main target application is subgraph queries between two (possibly singleton) sets of vertices V_S and V_T. Such queries are more efficient to implement using bidirectional breadth-first search, alternating between left-to-right (starting from V_S) and right-to-left (starting from V_T) iterations. This is because the search frontier, i.e., the set of states at a given path length, may grow very fast with path length (depending on the grammar and graph structure, exponentially in the worst case). Right-to-left searching is done the in the same way as left-to-right searching, except that the dot movement is from right-to-left in the dot conditions. We denote the state set in left-to-right and right-to-left direction with S_L and S_R, respectively.

Bidirectional search adds another layer of complexity: when left-to-right and right-to-left search frontiers collide in vertex v at path lengths j and j', respectively, we must check if there are pairs of states from $S_L(j,v)$ and $S_R(j',v)$ that, combined, form a valid parse for at least one path from V_S to V_T. Algorithmic details and proof of correctness of this operation are given in the Appendix.

4 Experiments

The goal of the experiments was to compare the performance of the proposed algorithms to each other and the baseline provided by an algorithm that enumerates all individual paths, as well as to verify their sufficient computational efficiency. Due to the novelty of the problem definition, we could not perform comparisons to existing methods. Randomly generated graphs were used for comparing the methods to each other in a controlled manner, while a large data warehouse derived from public biological databases was used to test their practical efficiency.

4.1 Data

A simple model was used to generate random graphs for the experiments: the number of vertices was fixed to 10000, and a varying number of undirected edges were added between randomly chosen pairs of vertices. The density of the graphs was varied by using 25000 to 250000 edges (in increments of 25000), which corresponds to probabilities between 0.0005 and 0.0050 for having an edge between a random pair of vertices. The type of each vertex and edge was drawn from a uniform distribution, where the number of edge and vertex types were 5 and 2, respectively. For each density, we generated 10 independent replicates.

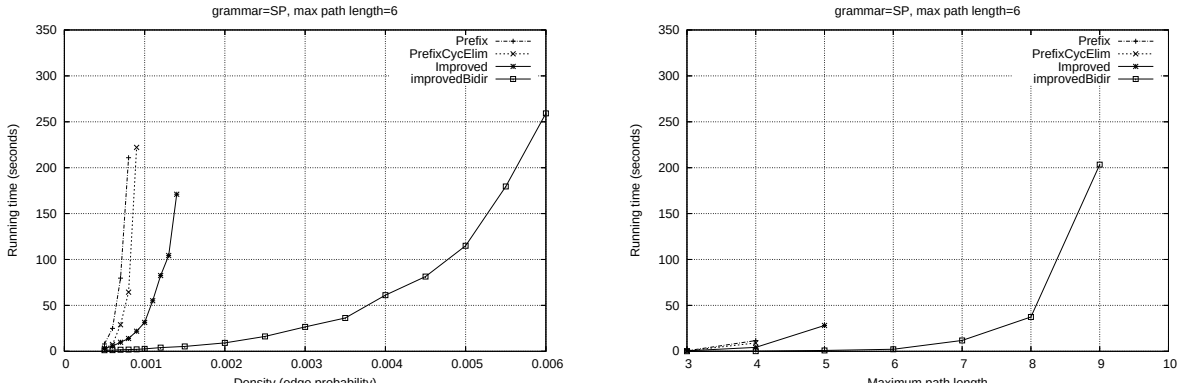
A real biological data set was obtained by downloading a set of publicly available data resources: UniProt (proteins), Entrez Gene (genes), Gene Ontology (protein functions, biological processes, and cellular locations), InterPro (protein families and conserved domains), KEGG (pathways), OMIM (gene-phenotype relationships), HomoloGene (gene homology groups), and STRING (protein interactions). The data from the source databases is transformed into a labeled graph representation, and stored into a local data warehouse. The data was restricted to a set of five organisms: Homo sapiens, Rattus norvegicus, Mus musculus, D. melanogaster and C. elegans. As a result of combining the data of the selected organisms from all the source databases, we get a graph consisting of approximately one million vertices and 5.7 million edges.

4.2 Comparison of algorithms with randomly generated graphs

The randomly generated graphs consist of five edge types `a,b,c,d,h` and two vertex types `Vm,Vn`. In these experiments, we used a grammar which represents a class of paths conferring similarity between the end-vertices. The grammar consists of a set of rules of form `E -> e V -e | h; V -> V E v | v`, where `e` is instantiated with edge types `a,b,c,d` and pair `(V,v)` is instantiated with vertex nonterminal–terminal-pairs ((`VN,Vn`),(`VM,Vm`)). Edge type `h` has a special role; an edge of type `h` indicates a direct similarity annotation between its end-vertices. The starting non-terminal of the grammar is `E`. The paths accepted by the grammar consists of nested palindromes, where the vertex-like non-terminal `V` can be expanded to a sequence of palindromes, with the addition of allowing a similarity edge `h` in the middle of the palindrome.

Four different parsing algorithms were tested. The trivial alternative is the prefix-tree algorithm (described in Section 3.1), which effectively enumerates all matching paths and returns their union. We implemented two variants of the prefix tree algorithm; PrefixCycElim with complete and Prefix with incomplete (equivalent to the other algorithms) cycle elimination. The other tested algorithms are the improved connection subgraph algorithm (Improved, Section 3.2) and its bidirectional variant (ImprovedBidir, Section 3.3).

All the algorithms were implemented using the Python programming language. Experiments were run on a standard PC with 1 GB of memory, running Linux. Each experiment was run separately for the the 10 independently generated graphs with same density, and we report average results over them. The number of states generated during the algorithm is directly proportional to the memory consumption of the algorithm. To avoid running out of memory, the the maximum number of states was fixed to 4 million; in practice this limit is always reached in less than approximately 6 minutes (when it is reached at all); in many settings, this limits the experiments with the more inefficient algorithms to small path lengths and/or densities.



(a) (b)

Figure 4: Average running times as a function of graph density (a) and maximum path length (b).

In each run of the algorithm, two end-vertices were randomly chosen, and a connection subgraph query was run with a specified maximum path length and the above-described grammar. Two factors were varied to control the difficulty of the parsing task: maximum path length of accepted paths and the density of the input graph.

We compared the algorithms to each other on randomly generated graphs with varying density (with maximum path length was fixed at 6) and varying maximum path length (between 3 and 10, with density fixed at 0.0015). Unlike in the Algorithms section, here path length is the number of edges in the original vertex and edge labeled graph; maximum path length n corresponds to maximum length $2n-1$ of open-ended path type (path length in the edge-only-labeled graph) for the parser. The average running times (over the 10 replicates) are shown in Figure 4. These experiments show significant differences between the algorithms; ImprovedBidir is clearly superior to the other algorithms, essentially allowing twice as long paths to be discovered as with Improved. The prefix-tree algorithms are clearly worst; however, it can be seen that PrefixCycElim is more efficient of these two. The line of each algorithm ends with the largest value of the varied parameter that the algorithm could be run with without exceeding the maximum state limit; e.g., with densities above 0.006, the ImprovedBidir algorithm would have exceeded the limit.

4.3 Tests with real data

To test the efficiency of the method on real data, we ran a series of experiments using our biological data warehouse. For this experiment, we randomly sampled 300 pairs of genes, and used the ImprovedBidir algorithm to find connections between each of these pairs. We used a grammar similar to the one used in the experiments with the random graphs (of course replacing the vertex and edge types with ones from the real database; the complete grammar is too large to describe here).

For each pair of genes, we ran the algorithm with three different maximum path lengths: 4, 6 and 8. Figure 5 illustrates the results of this experiment as a scatter plot; each point in the figure represents a single run of the algorithm, with result size (number of edges in the connection subgraph) on the x-axis and running time on the y-axis. Of course, in most cases there are no significant connections between a pair of randomly chosen genes. On the other hand, in cases

where there are short connecting paths between the genes, the returned connection subgraph may be very large, in which case it may be sensible to use a smaller maximum path length constraint. Therefore, the figure only shows runs where size of the result is within a reasonable range (1–200 edges); runs in which a connection does not exist are not displayed (in the cases where connections do not exist, the average running times were 0.3, 3.1 and 27.6 seconds for path lengths 4, 6 and 8, respectively). The results indicate that the algorithm works in reasonable time also with the real, large database and a relatively complex grammar.

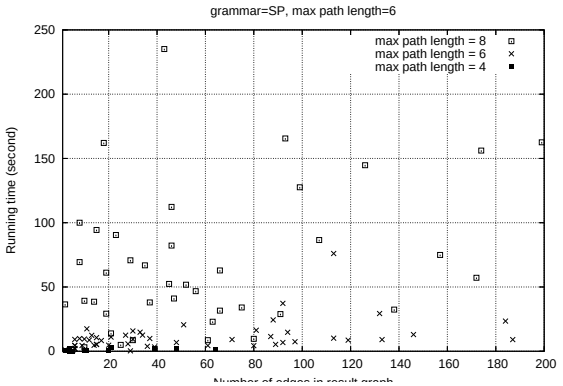


Figure 5: Running times with real data

5 Conclusions

We proposed a method for querying subgraphs of labeled graphs using a context-free grammar. Instead of returning all paths matching a query, our method returns a connection subgraph—the union of all matching paths. We presented algorithms that produce the connection subgraph directly, without first enumerating all the individual matching paths, and demonstrated its performance superiority on randomly generated graphs.

Tools and techniques for querying heterogeneous data sets are needed in various domains such as biology, social networks, WWW, and relational databases. Labeled graphs are a natural representation for many such data sets. Subgraphs connecting pairs of vertices represent complex relationships, and can reveal previously unknown links. The algorithms presented in this paper allow performing focused subgraph queries much faster than by using algorithms based on enumerating all paths matching the query. As a result, our algorithms are more suitable for interactive use; connection subgraphs induced by substantially longer paths can be extracted in acceptable waiting time.

The presented algorithms achieve their performance at the cost of incomplete cycle elimination. How big a problem this is depends on the application and characteristics of the data at hand. As the experiments in this paper indicate, the prefix-tree algorithm is practically too slow to parse paths longer than four edges. A similar speed-up to that seen with the improved algorithm can be achieved using a bidirectional search strategy, but still the relative difference to the bidirectional improved algorithm would likely be comparable to the difference between the prefix-tree algorithm and the improved algorithm. One possible solution to obtain cycle-free paths faster is to first use the bidirectional improved algorithm to produce an intermediate result

subgraph and subsequently eliminate cyclic paths using the bidirectional prefix-tree algorithm, which should perform much faster on the smaller intermediate graph.

Acknowledgments

Thanks to Hannu Toivonen for valuable comments and suggestions for improvements. This research has been supported by Tekes, Jurilab Ltd., Biocomputing Platforms Ltd., and GeneOS Ltd.

6 Appendix: Bidirectional parsing algorithm

In the bidirectional variant of the parsing algorithm, when left-to-right and right-to-left search frontiers collide in vertex v at path lengths j and j', respectively, we must check if there are pairs of states from $S_L(j, v)$ and $S_R(j', v)$ that, combined, form a valid parse for at least one path from V_S to V_T. This operation is described below.

Definition 1. (Parent states) States of form $p = (X \to \alpha \cdot A\beta, k, w) \in S_L(i, u)$ are the left *parents* of state $s = (A \to \delta \cdot \gamma, i, u)$, $p \xrightarrow{L} s$, i.e., states from which $(A \to \cdot\delta\gamma, i, u)$ is predicted. The right parents $p' \xrightarrow{R} s'$ are defined symmetrically.

For each generated state s_k (in either parsing direction), there is at least one parent-sequence $s_0 \dot{\to} s_1 \dot{\to} \ldots \dot{\to} s_k$ from an initial state s_0 to s_k. Note that a state may appear many times in such sequences (due to recursive productions such as $A \to Aa$).

Definition 2. (State compatibility) Compatibility is defined recursively: left-to-right state $s = (D, k, w) \in S_L(i, u)$ and right-to-left state $s' = (D', k', w') \in S_R(i', u')$ are *compatible*, $s \sim s'$, if and only if

1. for some nonterminals A and B, and some $\alpha, \beta : D = A \to \alpha \cdot B\beta$ and $D' = A \to \alpha B \cdot \beta$, and
2. either
   - A is the starting non-terminal Q, $i = i' = 0$, $u \in V_S$, and $u' \in V_T$, or
   - a compatible pair of parents $p \sim p' : p \xrightarrow{L} s, p' \xrightarrow{R} s'$ exists.

Compatibility of states s_k and s'_k means that there is a sequence of compatible state pairs $(s_0 \sim s'_0), (s_1 \sim s'_1), \ldots, (s_k \sim s'_k)$ from a compatible pair of initial states $(s_0 \sim s'_0)$ to $(s_k \sim s'_k)$ such that $s_0 \xrightarrow{L} s_1 \xrightarrow{L} \ldots \xrightarrow{L} s_k$ and $s'_0 \xrightarrow{R} s'_1 \xrightarrow{R} \ldots \xrightarrow{R} s'_k$. The latter (right-to-left) parenthood sequence can be viewed as a sequence of completions for the compatible states in the former parenthood sequence.

Theorem 1. *If states $s = (A \to \alpha \cdot B\beta, k, w) \in S_L(i, u)$ and $s' = (A \to \alpha B \cdot \beta, k', w') \in S_R(i', u')$ are compatible and a path in class B exists from u to u', then there is a valid parse for a path from V_S to V_T.*

Proof. The conditions effectively state that there is a path in class A from w to w'. If A is the starting non-terminal Q, $i = i' = 0$, $u \in V_S$, and $u' \in V_T$, then the theorem holds trivially. Otherwise, s and s' have a compatible pair of parents, p and p', and we can conclude by induction that the theorem holds. □

If path $\mathbf{p}$ of length k from V_S to V_T has a valid parse, then the search frontiers collide in some vertex v_0 on $\mathbf{p}$ at some left-to-right path length j_0 and right-to-left path length $k - j_0$. The following theorem states the necessary and sufficient conditions for existence of a valid parse for some path from V_S to V_T visiting vertex v_0 in case of collision at v_0.

Theorem 2. *Left-to-right state $s = (D, i, u) \in S_L(j_0, v_0)$ and right-to-left state $s' = (D', i', u') \in S_R(k - j_0, v_0)$ colliding at vertex v represent a successful parse, if and only if*

1. *the states have identical dot conditions ($D = D' = A \to \alpha \cdot \beta$), and*
2. *either*
   (a) *A is the starting non-terminal Q, $i = i' = 0$, $u \in V_S$, and $u' \in V_T$, or*
   (b) *a compatible pair of parents $p \sim p' : p \xrightarrow{L} s, p' \xrightarrow{R} s'$ exists.*

Proof. First, we prove that whenever the conditions are satisfied, a successful parse exists for a path from V_S to V_T. The first condition effectively states that there is a path in class A from u to u'. According to Theorem 1 this and compatibility of the states imply that there is a matching path from V_S to V_T.

Second, we prove that any path $\mathbf{p}$ of length k matching the query satisfies the conditions of the theorem. We denote the subpath from the ith to the jth vertex on path $\mathbf{p}$ by $\mathbf{p}[i, j]$. Either

1. there are states $(Q \to \alpha \cdot \beta, 0, u) \in S_L(j_0, v_0), u \in V_S$ and $(Q \to \alpha \cdot \beta, 0, u') \in S_R(k - j_0, v_0), u' \in V_T$, in which case conditions 1 and 2a are satisfied, or
2. there are states $s = (Q \to \alpha \cdot A\beta, 0, u) \in S_L(j, v), u \in V_S$ and $s' = (Q \to \alpha A \cdot \beta, 0, u') \in S_R(j', v'), u' \in V_T$ such that $j < k - j'$, $j \le j_0 \le k - j'$, and path $\mathbf{p}[j, k - j']$ is in class A.

In the latter case $s \sim s'$.

Suppose that, for some i and i', subpath $\mathbf{p}[i, k-i']$ is in class A, and states $p = (B \to \delta \cdot A\gamma, n, w) \in S_L(i, u)$ and $p' = (B \to \delta A \cdot \gamma, n', w') \in S_R(i', u')$ are compatible. Now either

1. there are states $(A \to \alpha \cdot \beta, i, u) \in S_L(j_0, v_0)$ and $(A \to \alpha \cdot \beta, i', u') \in S_R(k - j_0, v_0)$, or
2. there are states $s = (A \to \alpha \cdot C\beta, i, u) \in S_L(j, v)$ and $s' = (A \to \alpha C \cdot \beta, i', u') \in S_R(j', v')$ such that $j < k - j'$, $j \le j_0 \le k - j'$ and path $\mathbf{p}[j, k - j']$ is in class C.

In the former case, conditions 1 and 2b are satisfied. In the latter case, because $p \sim p'$, $p \xrightarrow{L} s$, $p' \xrightarrow{R} s'$ and the dot conditions of s and s' match, s and s' are compatible. By induction we conclude that some subpath $\mathbf{p}[i, k - i']$ always satisfies the former case, completing the second part of the proof. □

The parenthood relations form directed graph G_S over the coupled state-space $\bigcup_{j,v} S_L(j,v) \times \bigcup_{j,v} S_R(j,v)$: there is an edge from (s,s') to (p,p') if $p \xrightarrow{L} s$ and $p' \xrightarrow{R} s'$. Theorem 2 transforms directly into a depth-first search algorithm in G_S; state-pairs are labeled as compatible or incompatible during backtracking according to the conditions stated in the theorem. The algorithm uses dynamic programming: since the label of a state-pair only depends on its descendants in the search space, there is no need to continue the search further from an already labeled state-pair. For ill-defined grammars (e.g., one containing productions $A \to B, B \to A$), G_S may contain cycles. The algorithm can be applied to such grammars; it eliminates such cycles by not revisiting unlabeled state-pairs (i.e., state-pairs on the current search path). If state-pair $(s,s') \in S_L(i,u) \times S_R(i',u')$ is deemed compatible, states s and s' are added to their respective sets of accepted states, $A_L(i,u)$ and $A_R(i',u')$.

The algorithm outlined above does not as such address partial elimination of cycles emerging from combining the partial parses. Such cycles can be eliminated by modifying the algorithm to check each state-pair for cyclicity. State-pairs failing the acyclicity test are left unlabeled; a subsequent test for the same pair might pass the test.

The backtracking phase can be done in the same way as in the unidirectional algorithm, but separately for each direction, starting from the respective set of accepted states, $\bigcup_{j,v} A_L(j,v)$ or $\bigcup_{j,v} A_R(j,v)$.

References

[1] L. Getoor and C. P. Diehl. Link mining: A survey. *ACM SIGKDD Explorations Newsletter*, 7(2):3–12, 2005.

[2] M. P. Consens and A. O. Mendelson. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404–416, New York, NY, USA, 1990. ACM Press.

[3] S. Abiteboul and V. Vianu. Regular path queries with constraints. *Journal of Computer and System Sciences*, 58(3):428–452, 1999.

[4] S. Flesca and S. Greco. Querying graph databases. In *Proceedings of EDBT 2000: 7th International Conference on Extending Database Technology*, pages 510–524, London, UK, 2000. Springer.

[5] Z. Lacroix, L. Raschid, and M.-E. Vidal. Efficient techniques to explore and rank paths in life science data sources. In *Proceedings of Data Integration in the Life Sciences, First International Workshop (DILS 2004)*, pages 187–202, 2004.

[6] A. O. Mendelson and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24:1235–1258, 1995.

[7] P. Mork, R. Shaker, A. Halevy, and P. Tarczy-Hornoch. PQL: A declarative query language over dynamic biological schemata. In *Proceedings of the American Medical Informatics Association Annual Symposium 2002*, pages 533–537, 2002.

[8] U. Leser. A query language for biological networks. *Bioinformatics*, 21(Suppl 2):ii33–ii39, 2005.

[9] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 323–330, New York, NY, USA, 1987. ACM Press.

[10] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.