Algorithms for the Compression of Graphical Data

This section defines the formal problem of compressing a dataset of graphical traces, assumed to have been generated from some underlying graphical process model, *M*. The ability to induce the graphical structure of a process from trace data may seem of limited application outside of process mining, business process management, or similar operations fields. However, the induction of graphical structure from data encompasses many classical formal problems spanning machine learning, planning, and artificial intelligence, as will be discussed.

Consider a planning problem of a robot navigating two-dimensional space, using elementary reinforcement-learning formalisms. Assume the action set contains four discrete actions , the environment is deterministic and known, the transition model may be known or unknown, and the robot’s task is to navigate to some positive reward location while avoiding obstacles with negative rewards. These problem formulations are amenable to a wide variety of approaches in a class of algorithms modeling the sequential one-step dynamics of discrete action sets, for relatively modestly sized action sets. Many canonical solutions are rooted in Bellman equation formalisms or Monte Carlo methods, an overview of which is provided by [Sutton and Barto (1998)].

Now instead consider that the rewards and system dynamics are determined by *k*-step bounded dynamics, where the robot achieves goals only by executing specific partially-ordered sequences of actions of length *i*, for . For example, or ; additionally these action structures may be interspersed with actions for unrelated tasks. These representations mimic real-life tasks, where actions possess long-range interdependencies, and tasks recursively decompose to subsets of actions, or “subtasks”, which do not necessarily entail one another despite their adjacency. Moreover, such tasks often can only be described graphically, by process model formalisms such as Petri Nets [Peterson, 1981].

These structured environments violate the clean, Markovian one-step sequential dynamics required by many classical reinforcement learning formulations, and result in exponential search complexity. For instance, the long-range activity set of action sequences in the above example is exponential in the number of activities, , and the bound on sequence lengths, giving . This is an intractable space even for this modest activity set, and without any assumptions about the complexity of the action space, which is inherently non-sequential.

These problems are the domain of classical planning problems, such as the block world domain of [Nilsson, 1980], for which a complexity analysis is provided by (Gupta et al, 1992). More accessible examples are the games of Go and Chess, with their high branching factor and multiple game strategies composed of long-range action dependencies. In these problems, the action space, action-sequence space, state space, or combinations of these are intractably large for traditional, sequential learning formulations without search heuristics. Current approaches often implement approaches such as Monte Carlo Tree Search [Brown et al., 2012], which still entail intensive search behavior before upper confidence bounds on action-value estimates yield satisfactory performance.

Graph compression addresses these problems heuristically by extracting the subgraphs characterizing advantageous system behavior. For instance, in a highly complex combinatorial space such as chess, sub-graphs of useful actions can be learned via user examples. Likewise, non-adversarial everyday tasks such as performing home chores or driving may be learned from examples characterizing the underlying graphical model of the task. Given large amounts of such user data for this and similar planning problems, repeated subgraphs can be extracted and used to bootstrap learning algorithms to bias their activities toward advantageous structural features. Conversely, the extracted information can be used for tasks such as anomaly detection.

Thus, many planning problems can be solved heuristically by learning graphical features of processes. These representations can then be used to efficiently learn complex behavior from compositions of subgraphs representing subtasks within a domain. As a result, methods for compressing and extracting structural patterns from graphical data have general application in machine learning, planning, and artificial intelligence, and encompass general task-learning.

The Optimization Problem of Graphical Data Compression

In this work, we wish to compress a set of graphical data representing directed, possibly-cyclic traces generated from an underlying graphical model, . In the domain of process mining, these models typically have regular, compression-favorable structural properties, such as defined *begin/entry* and *end/exit* points for traces, modest overall average degree, few or zero cycles, and generally, if not always, behaving like directed, acyclic graphs (DAGs). The field of process mining provides many algorithms for mining from some input log ; can then be used to convert a log of partially-ordered traces into a set of corresponding graphical traces (hereafter simply ).

The goal of graphical compression is to reduce a set of graphical traces to a minimal collection of prototype subgraphs , or edge subsets, maximally compressing :

This construction searches for the minimum *k-*bit length set, , given log , although other criteria could be devised. Once the set is found, each trace can be encoded as a *k*-length binary vector indicating its member subgraphs, and thus the trace can be encoded and decoded via and . Thus, a lossless compression method can be devised by finding the minimal , converting each trace to a bit vector indicating its subsets in , and transmitting these vectors along with the edge-subsets by which to decode them.

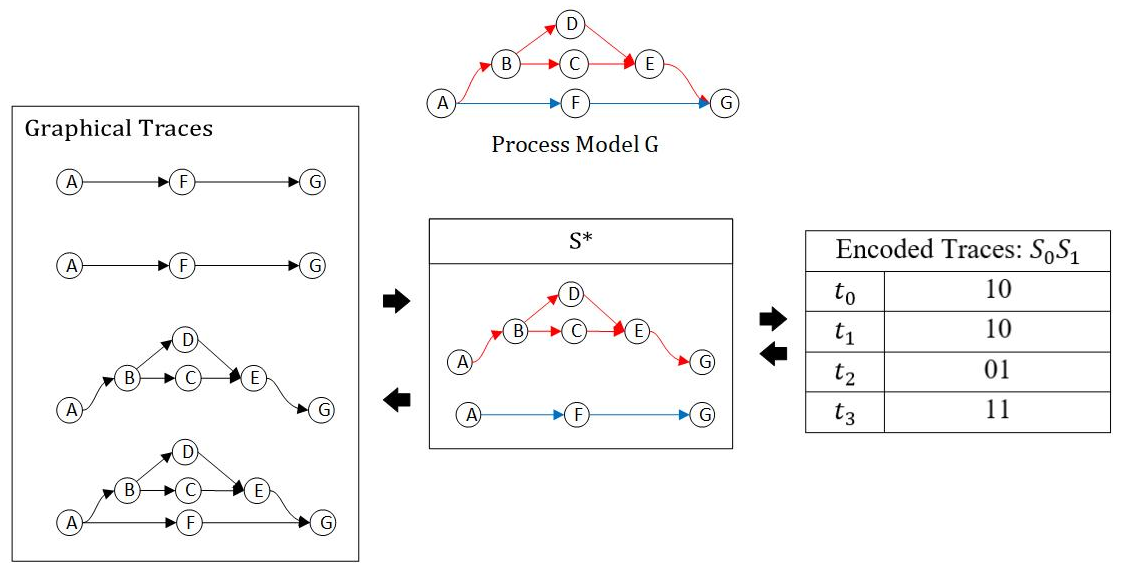


Figure : Graphical encoding and decoding. Trace set 'T' is shown at left, compressing set S\* (center), and the encoding vectors for each of the four traces (right), where a '1' indicates Si is present. S\* provides the encoding/decoding dictionary.

The general method is illustrated above in figure 1. Assume that any trace only executes one (or both) of the colored paths shown as the process graph . Then the colored components form substructures by which the entire log of traces may be compressed (encoded), by converting each trace to a binary vector. The colored substructures provide the mapping to and from these vectors.

This is also a very simple instance of graph grammars and graph parsing. Figure 1 provides a direct example of a simple graph parsing output, where the encoded binary vector high-bits flag the production rules by which to convert any trace back into its graphical form via the production rule set *S*. The similarity is coincidental, since graph grammars encompass more complex recursive rules than the example in figure 1. However, many graph grammar induction algorithms have been published, given their relevance to tasks such as compiler construction and various mathematical problems. An example of heuristic graph grammar induction is given by [Jonyers et al., 2002] using a similar approach to the one demonstrated later in this work for dendrogram-induction from process data.

A Naïve, Illustrative Problem Formulation:

The formal problem of finding the compressing set reduces to iteratively selecting columns from the unfolded binary adjacency matrices of all traces in the directed graphical data. For a dataset in the form of a trace log, , of size , each trace is a subgraph of the super-graph . represents the graphical behavioral model inclusive of all traces, which is the union of all edges in . Each trace can be represented as an adjacency matrix whose rows and columns are composed of the vertices of . Concatenating the rows of this matrix gives a binary vector whose non-zero components indicate the directed relations (edges) present in the trace, as shown in figure 1.

This representation is solely for elaboration purposes, since this binary representation only captures first-order structure. For instance, it does not quantify multiple loop executions. However, it is sufficient to describe many graphical structural properties via labeled, directed graphs with unweighted edges.

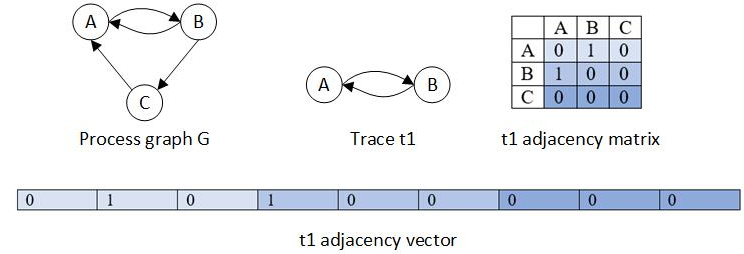


Figure 2: Shown above are the graph G, the process model inclusive of all traces. Trace t1 is an example subgraph of the execution of this process, with its associated adjacency matrix and adjacency vector representation.

The entire set of traces can be converted to a matrix, of adjacency ‘vectors’ like the one shown above in figure 1. A hypothetical matrix is shown below in figure 2, whose shaded regions are explained further below. This data representation is illustrative because it demonstrates the dimensionality of the input space, which is quadratic in the number of vertices, for a complete, directed graph *G*.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Edges, | | | | | | | | | | |
| Traces |  | A-A | A-B | A-C | B-A | B-B | B-C | C-A | C-B | C-C |
| t1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| t2 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| t3 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| t4 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| t5 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| t6 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

Figure : A log data representation in which each trace’s adjacency matrix is unfolded into a row. The complete log includes all such rows.

Discovering the maximally compressing subgraph within this data representation reduces to finding the largest subset of columns containing all 1’s and encompassing the greatest number of rows. Such a column set of edges represents a subgraph , if connected. The size of the column set represents the size of the subgraph in edges, likewise, the number of rows is its frequency.

In this manner, a maximum-compressing subgraph can be found at each iteration by searching the columns of this matrix for the largest collection of all 1’s. The shaded columns in figure 3 depict such a search; the highlighted rows are rows containing all 1’s for the given column selection. For any fixed choice of columns, all traces must be traversed to count the number of rows for which the conjunction of a fixed choice of columns evaluates to true, and there are possible choices of columns. Defining the set of all edges in graph as , the set of all vertices and using the identity (Cormen et al, 2001), the complexity of this procedure is given by:

Equation : Search complexity

Thus, this column selection procedure is exponential in the number of edges, which in the worst-case for a directed, fully-connected, and non-reflexive graph, is:

Hence,

Equation : Bounded search complexity

This search procedure illustrates the problem’s structure and brute force complexity for only a single compression iteration. But it is flawed since it needlessly searches over all combinations of subsets of edges of size *i*, whereas we are only interested in connected components of the . Graphical problems frequently involve sparse graphs, so we can expect to reduce complexity by restricting iterations to the subsets of columns representing connected components. For special kinds of graphs like DAG’s, components can be enumerated using basic, elementary graph-search procedures. Lastly, real graphical data typically has high redundancy, such that can also be reduced significantly by a de-duplication strategy of storing each unique row with its frequency, and further by omitting columns of all zeroes, such as the ‘C-B’ and ‘C-C’ columns in figure 2.

Critically, this search characterization overlooks how subgraph size and subgraph frequency affect the optimality of the resulting set . The search over columns (substructures) and rows (frequency) introduces a compression tradeoff between size and frequency: it may be easy to find a very large and infrequent subgraph, or conversely a very small but frequent subgraph (such as a single edge). Consider two candidate prototypes, and . Let have and , and let have and . Which of or should be chosen to obtain optimal compression, ? How do criterions of and affect the optimality of the resulting set ? A straightforward heuristic is to select the subset with the largest sum of 1’s as a measure of information gain.

Worse yet, since candidate subgraphs for compression are not disjoint, dependencies exist between their selection. The selection of a prototype at the t*th* iteration can affect which candidates are available in subsequent iterations, which may affect the compression of the resulting set . Optimal compression is defined as minimizing the description length of the trace-graph codes sufficient to losslessly decode all trace-graphs from their encodings via . Due to these tradeoffs, maximally encoding the subgraphs requires correctly making the complete sequence of decisions per the size and frequency of each prototype subgraph. This problem is akin to bin-packing [Korte, 2008], an NP-hard combinatorial problem, but harder due to the overlapping dependencies between prototype selection. Loosely, each prototype’s size and frequency define the object’s abstract dimension, while the objective is to fit as many of these objects as possible into the smallest bin. The similar sequential problem of generating the smallest context-free grammar generating a sequence of symbols (e.g. traces) is also known to be NP-hard [Charikar et al., 2005]. Fortunately, the optimal formulation of this problem is not the subject of this work.

The Heuristic View

In a relaxed version of the trace-graph compression problem the encoding need not be optimal but may instead heuristically generate an approximately optimal encoding much faster than the optimal . Such procedures can still be lossless, such that any trace can be completely reconstructed from its encoding and the set to decode it. Many heuristics are possible, since the task is relaxed to that of iteratively finding frequent subgraphs under some compression criterion, such as favoring the size or frequency of graphs. Many of these heuristics can be viewed as greedy methods for reducing the column selection search or for reducing the leading constant in equation 2. An overview of such methods can be found in [Maneth et al., 2017]. Many graph problems involve graphs with node attributes or other additional information that may also be incorporated into the information theoretic definition of their encoding.

The prior problem of discovering the smallest context-free grammar for generating sequences relates to graph grammars and graph parsing. The example in figure 1 provided a direct example of a simple graph parsing output, where the encoded binary vector high-bits flag the ‘rule productions’ by which to convert any trace back into its graphical form via the production rule set *S*. The similarity is coincidental, as graph grammars normally encompass complex recursive rules. However, many graph grammar induction algorithms have been published, given their relevance to computing tasks such as compiler construction and various mathematical problems. An example of heuristic graph grammar induction is given by [Jonyers, 2002] using a similar approach to the one demonstrated later in this work for dendrogram-induction from process data.

Notably, the adjacency-matrix based data representation in the prior section is amenable to a wide range of supervised and unsupervised learning approaches. Unsupervised approaches, such as neural autoencoders, possess great promise in terms of automating graphical pattern discovery. This framework trains a neural network using the input as the target output. Using various training and architectural strategies, these networks learn the hidden structural patterns of the data, by which normative and anomalous patterns can be determined. A recent example is given by [Nolle et al, 2016], in which the authors used a denoising autoencoder model for both anomaly detection and normative pattern discovery. The authors presented the traces to their networks as linear activity sequences, which is the canonical process mining trace representation, rather than trace adjacency matrices. They reported their method perfectly split the trace log into normal and anomalous traces. Similar work is possible using recurrent neural networks [Ellman, 1990] [Jordan, 1986] by presenting the traces to the network as linear activity sequences, although the hidden layers of such a model would be difficult to interpret. More recent neural models combine recurrent networks and autoencoders for a variety of similar structured input/output tasks, such as the seq2seq model of [Kalchbrenner et al., 2013] or the long-short-term memory (LSTM) autoencoder framework described in [Srivastava, 2015].

Supervised learning models can also be adapted to unsupervised pattern discovery. By appending a +1 to each binary adjacency ‘vector’ as a dummy target “output” for a learning model, the unsupervised data can be mapped to a supervised learning representation. Each vector can likewise be replicated by its negation, possibly with additive noise, to generate a semi-synthetic supervised-learning dataset dividing the input space into two classes: positive examples, and ‘negative’ synthetic examples sampled outside the set via some distribution facilitating a specific learning model. Some distant examples of such data extension/generation strategies are the negative sampling used by some implementations of the word2vec algorithm (Mikolov et al, 2013), various structured learning algorithms like the DAgger algorithm (Bagnell, 2015), or (distantly) the generative adversarial networks of (Goodfellow et al, 2014).

The benefit of such a supervised data representation is that many supervised learning models have been developed, especially generative ones, by which normative patterns or other model parameters can be learned to perform secondary tasks like anomaly detection and normative pattern extraction. The simplest approach is to run linear regression on the preceding semi-synthetic data description. The result output is a weight vector whose non-zero components correspond to the collection of edges which maximally “compress” the data by minimizing the mean-squared error (MSE) loss. The corresponding columns would then be removed from the input data, and the procedure would be re-run on the remaining examples to find the next set of such edges, and so on, until the data is completely compressed. Notably, the edge collection found on any iteration might not represent a connected subgraph, but regularization strategies could be devised to bias the learning algorithm toward connected, rather than disconnected edge subsets.

SUBDUE

While many data representations and strategies are left to be explored, this work’s primary focus is on the SUBDUE method for discovering the maximally compressing components of graphical input data. In contrast to matrix representations of graphical data, SUBDUE is search-based and focuses on the vertex perspective to search for compressing substructures. In this manner, SUBDUE proceeds by “growing” candidate substructures within some search beam of size *k*, maintaining only the most-highly compressing components in the beam at any time. Compression is measured by the reduction in the description length of the data with respect to compressing components, via the minimum-description length (MDL) principle [CITE].

From this perspective, compressing components are found not by solving a brute-force global search over edges, but rather by growing compressing components from the neighborhood surrounding promising nodes.

SUBDUE fits neatly into these purposes, since the algorithm compresses not just based on the frequency of a subgraph, but also some metric of its encoded length…

Nilsson, N. J. (2014). *Principles of artificial intelligence*. Morgan Kaufmann.

Gupta, N., & Nau, D. S. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, *56*(2-3), 223-254.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction* (Vol. 1, No. 1). Cambridge: MIT press.

Peterson, J. L. (1981). Petri net theory and the modeling of systems.

Browne, Cameron B., et al. (2012). *A survey of monte carlo tree search methods*. IEEE Transactions on Computational Intelligence and AI in games 4.1 (2012): 1-43.

Bagnell, J. A. (2015). *An invitation to imitation* (No. CMU-RI-TR-15-08). CARNEGIE-MELLON UNIV PITTSBURGH PA ROBOTICS INST.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Introduction to algorithms second edition. p.102.

Korte, B., & Vygen, J. (2008). Bin-packing. *Combinatorial Optimization: Theory and Algorithms*, 449-465.

Nolle, Timo Unsupervised Anomaly Detection in Noisy Business Process Event Logs Using Denoising Autoencoders (2016)

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).

Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., & Shelat, A. (2005). The smallest grammar problem. *IEEE Transactions on Information Theory*, *51*(7), 2554-2576.

Maneth, S., & Peternek, F. (2017). Grammar-Based Graph Compression. *arXiv preprint arXiv:1704.05254*.

Kalchbrenner, N., & Blunsom, P. (2013). Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (pp. 1700-1709).

Srivastava, N., Mansimov, E., & Salakhudinov, R. (2015, June). Unsupervised learning of video representations using lstms. In *International conference on machine learning* (pp. 843-852).

Elman, Jeffrey L. (1990). "Finding Structure in Time". Cognitive Science. **14** (2): 179–211.

Jordan, Michael I. (1986). "Serial Order: A Parallel Distributed Processing Approach".

Jonyer, I., Holder, L., & Cook, D. (2002, July). Concept formation using graph grammars. In *Proceedings of the KDD Workshop on Multi-Relational Data Mining* (Vol. 2, pp. 19-43).

|  |  |
| --- | --- |
| Encoded Traces: | |
|  | 10 |
|  | 10 |
|  | 01 |
|  | 11 |