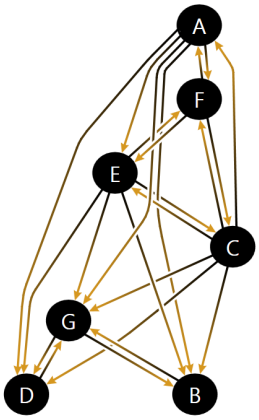
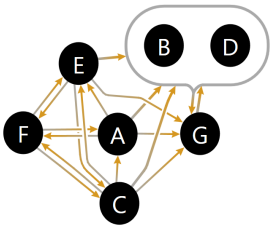
Edge Compression Techniques for Visualization of Dense Directed Graphs

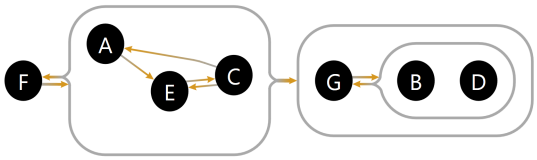
Tim Dwyer, Nathalie Henry Riche, Kim Marriott, Christopher Mears



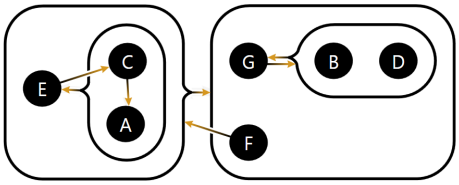
(a) A standard "flat" node-link diagram of a graph with 23 directed edges.



(b) Since B and D have exactly the same sets of neighbors they can be grouped leaving 18 edges using a simple *Neighbor Matching*.



(c) A *Modular Decomposition* allows internal structure within modules and nesting. Nine edges remain.



(d) A *Power-Graph Decomposition* further relaxes the definition of a module to allow edges to cross module boundaries, allowing the same graph to be drawn with only 7 edges.

Fig. 1. Different edge-compression techniques applied to the same small graph.

Abstract— We explore the effectiveness of visualizing dense directed graphs by replacing individual edges with edges connected to "modules"—or groups of nodes—such that the new edges imply aggregate connectivity. We only consider techniques that offer a lossless compression: that is, where the entire graph can still be read from the compressed version. The techniques considered are: a simple grouping of nodes with identical neighbor sets; *Modular Decomposition* which permits internal structure in modules and allows them to be nested; and *Power Graph Analysis* which further allows edges to cross module boundaries. These techniques all have the same goal—to compress the set of edges that need to be rendered to fully convey connectivity—but each successive relaxation of the module definition permits fewer edges to be drawn in the rendered graph. Each successive technique also, we hypothesize, requires a higher degree of mental effort to interpret. We test this hypothetical trade-off with two studies involving human participants. For Power Graph Analysis we propose a novel optimal technique based on *constraint programming*. This enables us to explore the parameter space for the technique more precisely than could be achieved with a heuristic. Although applicable to many domains, we are motivated by—and discuss in particular—the application to software dependency analysis.

Index Terms—Directed graphs, networks, modular decomposition, power graph analysis.

1 Introduction

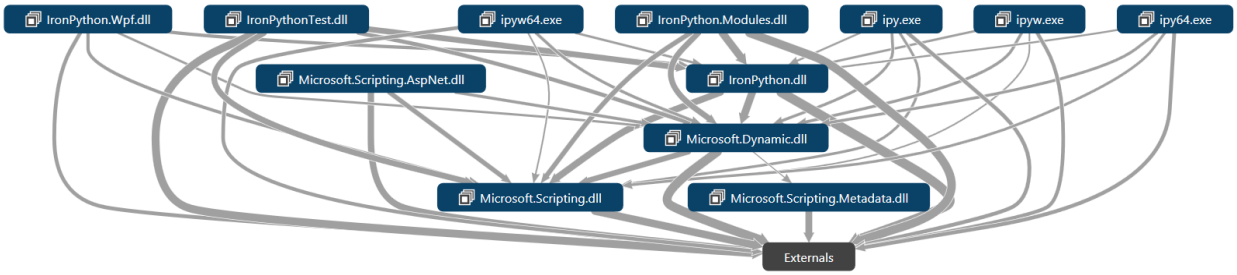
Tremendous work has been focused on the visualization of graphs, i.e. data structures composed of nodes connected by edges. This work has yielded techniques allowing the display of a large number of elements. However, the usefulness of these representations is doubtful when the graphs are so dense with edge curves as to be unreadable. This problem is made even worse when the directionality of the edges is important. Understanding the direction of dependencies in software-engineering diagrams or the direction of flow in biological reaction networks is essential to those applications and countless others. In such applications, to answer questions about directed connectivity each individual edge path must be traceable and its direction obvious.

We encountered this specific problem when developing a software dependency visualization feature for the Microsoft Visual Studio Integrated Development Environment. The tool's default behavior is to produce a diagram of the dependencies between the top-level components in a given piece of software using a node-link representation. The idea is to produce a relatively small overview of the system from whence the user can perform a top-down visual exploration into the full dependency graph associated with each component. However, for any but the most trivial systems, this top-level diagram may already be so dense with dependency links as to be difficult to read.

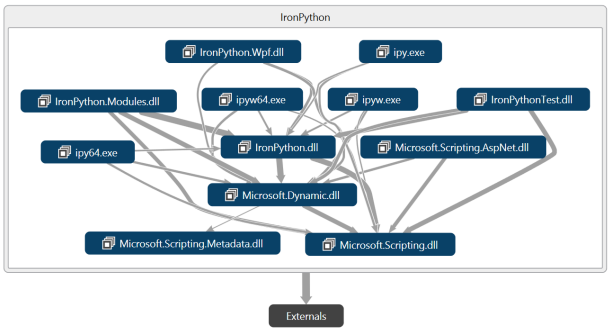
For example, Fig. 2(a) is the top-level graph produced by the tool for IronPython, an open-source .NET based implementation of the Python language. Part of the problem here is that the "Externals" node

- *Tim Dwyer is with Monash University, E-mail: tim.dwyer@monash.edu.*
- *Nathalie Henry Riche is with Microsoft Research, E-mail: nath@microsoft.com.*
- *Kim Marriott is with Monash University, E-mail: kim.marriott@monash.edu.*
- *Christopher Mears is with Monash University, E-mail: chris.mears@monash.edu.*

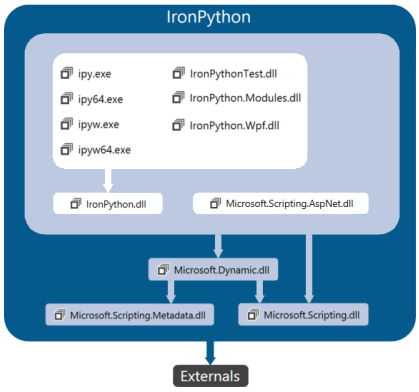
Manuscript received 31 March 2013; accepted 1 August 2013; posted online 13 October 2013; mailed on 4 October 2013.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.



(a) The top-level component graph produced by the Visual Studio code-dependency analysis tool for the IronPython code base with 39 edges.



(b) We remove the links from every element to the "Externals" node and replace them with a single link from the group of all nodes. The same information is presented but 12 links have been removed and one aggregate link added.



(c) A full modular decomposition of the same graph leaves only six aggregate links without any loss of connectivity information for an 85% compression rate.

Fig. 2. Edge compression techniques applied to the graph of dependencies between libraries in the IronPython [4] codebase. The representation in (c) starts to look like the kind of "poster" that makers of complex software frameworks sometimes produce to convey the high-level architecture of the platform, e.g. [13]. However, such poster views are usually conceptual and abstract. This representation is a precise depiction of the underlying code dependencies and can be generated automatically.

has a link from every other node in the graph. This means that every component in the system references a library external to the loaded code-base. Unfortunately, this is a common situation in software engineering since most code depends on other platforms. We cannot remove the "Externals" node since in this interactive system it provides a useful UI affordance: users can expand the node to see exactly which external libraries are referenced. We could instead adopt the convention of placing a box around nodes that are internal to the IronPython source-code. Then, one thick arrow from the box to the "Externals" node, as in Fig. 2(b), *implies* that every node inside the box has a link to the "Externals" node. This first transformation is fairly obvious, however, it turns out that by repeatedly applying the same type of grouping we can go significantly further to reduce the clutter, as in Fig. 2(c).

The only caveats to such edge-compression techniques are that the convention of links being "rolled-up" between groups needs to be learned and that people may have some difficulty in mentally "unrolling" in order to perform useful analysis. We argue in this paper that the benefits provided by edge-compression techniques outweigh these caveats.

Contributions. We examined three types of edge-compression techniques called *Matching Neighbors*, *Modular Decomposition* and *Power Graph Analysis*, extending the latter to handle directed graphs. We experimentally tested whether people with little graph expertise can learn to read such representations depending on the type and degree of edge compression. To our knowledge, we report the first empirical studies involving human participants on the readability of these techniques. Our results reveal their potential and immediate application to the visualizations of directed graphs in a variety of domains. Our first study, comparing flat directed graphs with graphs compressed using matching neighbors or modular decomposition, is presented in §4.

We then move beyond the simple techniques above to a generalization of edge compression as an optimization problem that we believe is NP-hard. This allows us to test different goal functions controlling the degree of compression. A prior technique *Power Graph Analysis* [23] attempted only to remove as many edges as possible. We hypothesize that graphs with such maximal compression may prove too difficult to read and that other objectives should be considered. However, a readability study using a heuristic compression may be significantly compromised by the limitations of the heuristic. We introduce a novel methodology involving the use of *Constraint Programming* to generate a corpus of *optimal* solutions to small compression problems with a range of objectives. To summarize, the contributions of this methodology are: a constraint optimization model, §5.1; the corpus of graphs and their optimal compressed solutions ready for other researchers to use, §6; the first empirical comparison of the power graph heuristic to the optimal, §7; and the results of our own controlled experiment on readability, §8.

2 Background

People have sought to provide simpler views of large and complex graphs by automatic grouping for some time. Most commonly this is done with techniques that elide information to provide a more abstract or high-level view. For example, Gansner *et al.* [12] use a spatial clustering based on an initial layout of a large graph to provide an extremely abridged overview of the full graph's structure. Abello *et al.* [5] compute a hierarchical clustering based on the graph structure that users are then able to navigate interactively. Note that both of these techniques remove both nodes and edges to produce their abridged views. A past evaluation of such a *lossy* graph compression technique is provided by Archambault *et al.* [6]. Their study evalu-

ates interactive navigation of clustered graphs. In this paper we focus on techniques that allow the precise structure of the entire graph to be inferred by the reader without the need for interaction.

To the best of our knowledge, the first work that used a grouping over nodes to allow implicit edges to be removed is the Phrase Nets text visualization system by van Ham *et al.* [24]. Before displaying the Phrase Nets they would find sets of nodes with identical neighbor sets. In the final drawing such nodes were grouped together in the diagram and the edges entering or leaving this group were understood to imply individual edges to each node in the group. Note that since these groups of nodes had to have identical neighbor sets there could not be any internal structure within the group (apart from cliques with common external neighbors which is an easily detectable special case). As we shall see in Section 3, this significantly limits the degree of edge compression that can be achieved in practice.

This simple edge compression technique has probably been reinvented many times. For example, Dinkla *et al.* [9] used a similar technique for grouping nodes with identical neighbor sets to produce compacted matrix representations of biological graphs.

The edge compression technique used in Fig. 2(c) is called a *Modular Decomposition*. A modular decomposition identifies groups or *modules* of nodes that have identical *external* connections. This definition of a module places no restrictions on the connectivity structure within the module. Therefore, modules may have internal edges and the decomposition can be applied recursively to obtain nested modules. The modular decomposition of a given graph is unique and it can be found in time linear in the number of nodes and edges [17].

To our knowledge, modular decomposition has not previously been used in graph visualization to avoid drawing the full set of edges. However, it has been used to assist with layout. Papadopoulos and Voglis [21] used Modular Decomposition as a means of obtaining layout for undirected graphs that considers the module structure. They used a force-directed technique to keep nodes in the same module physically close together. However, their final rendering was a typical node-link drawing with all edges present. An earlier layout method that employed similar ideas for drawing directed graphs was by McCreary *et al.* [18]. They looked for an alternative to the most commonly used hierarchical graph drawing technique using a so-called *Clan* decomposition. Similar to [21], they used a decomposition to achieve a more structured layout, but they still drew the full set of edges.

Modular Decomposition should not be confused with *maximal modularity clustering* [20]. The latter can be seen as another *lossy* compression technique. It attempts to group a graph into communities/groups/clusters/modules with minimal links between them compared to a random graph with similar degree distribution. Modularity maximization is NP-hard [8] and the optimal partitioning is not necessarily unique.

Power graphs were recently introduced by Royer *et al.* [23] for the simplification of dense undirected biological graphs. We describe in the next section how they can be thought of as a relaxation of the constraint in modular decompositions that the aggregate edges do not cross module boundaries. This relaxation allows for potentially much greater compression of edges. We investigate in this paper—for the first time—whether this comes at the expense of more difficult interpretation. Also, their heuristic was designed purely to minimize the number of edges in the decomposed drawing. In this paper we investigate the effect of optimizing other criteria.

To our knowledge none of the above techniques have been evaluated with controlled studies.

3 Definitions and Techniques

We consider three different edge compression techniques: *matching neighbors* as per [24, 9]; *modular decomposition* and *Power Graph Analysis*. In all cases we apply these techniques to a graph $G = (V,E)$ with node set V and directed edges E. A node $v \in V$ has incoming neighbors $N^-(v) = \{u \mid \exists(u,v) \in E\}$ and outgoing neighbors $N^+(v) = \{w \mid \exists(v,w) \in E\}$.

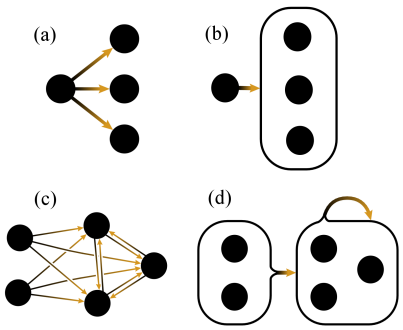


Fig. 3. The simplest Matching Neighbors compression allows for leaves of "star graphs" (a) to be collapsed into a module (b). A simple extended matching allows for cliques as in (c) to also be grouped (d). In (d) the edges in the clique module are removed and implied by a self-loop: an arrow from the module to itself.

3.1 Matching Neighbors.

The simplest definition for a module that provides a useful edge compression is that all nodes in a module must have identical neighbor sets. That is, two nodes u and v are grouped if $N^+(u) = N^+(v)$ and $N^-(u) = N^-(v)$. This allows for grouping of the leaves of stars (e.g. Fig. 3(a-b)) or for k-partite components into k modules.

Further compression is possible by relaxing the definition further to allow cliques with identical external neighbors to also be grouped into modules [24, 9], as in Fig. 3(c-d). That is, in addition to the above module definition they also allow two nodes u and v to be grouped if $u \in N^+(v), v \in N^+(u), N^+(u) \setminus \{v\} = N^+(v) \setminus \{u\}$ and the same is true for the incoming neighbor sets of u and v.

Simple matching in this way is trivially computed in linear time by hashing. Fig. 1(b) gives a non-trivial example of matching neighbors. For a given graph, the neighbor matching is unique.

3.2 Modular Decomposition.

The rules described above for matching neighbors do not allow for internal structure within modules: the nodes inside each module are either not connected to any other nodes in the module, or they are connected to every other node in the module. Modular decomposition relaxes this definition. Thus, a set of nodes $M \subset V$ is a module in a modular decomposition, if for every pair of nodes $u, v \in M$, $N^+(u) \setminus M = N^+(v) \setminus M$ and $N^-(u) \setminus M = N^-(v) \setminus M$. This definition allows for internal structure within modules, and therefore the possibility that the decomposition can be applied recursively to obtain nested modules. Examples of modular decomposition are given in Figures 1(c) and 2(c). Note that matching neighbor sets are subsumed by modules, so Figures 3(b) and (d) also happen to be modular decompositions.

The first polynomial-time algorithm for modular decomposition is due to James *et al.* [16]. Methods that require linear time in the number of nodes and edges exist for directed graphs [17] but they are subtle and difficult to implement. The figures in this paper were generated with an $O(|V|^2)$ method due to Ehrenfeucht *et al.* [11].

3.3 Power Graph Analysis.

Note that in all of the above definitions for modules, edges in the decomposition and hence the drawing, never cross module boundaries. Fig. 1 shows that greater edge compression can be achieved if the definition of a module is relaxed to allow edges to cross module boundaries. However, now there are many possible choices for how to decompose the graph in order to compress edges. Royer *et al.* [23] very briefly described a simple greedy heuristic for performing this so called *power graph analysis* for undirected graphs.

They begin by computing a similarity coefficient for all pairs of nodes based on their neighbor sets. Then they apply a hierarchical clustering of the nodes based on this similarity. Every cluster in the resulting hierarchy is considered as a candidate module.

The second part of the heuristic involves calculating, for each candidate module, the number of edges that could be removed were the module to be created. In a greedy fashion, they then create the module that allows the most edges to be removed and repeat until no further improvement is possible.

We consider Power Graph Analysis in more detail in §5.

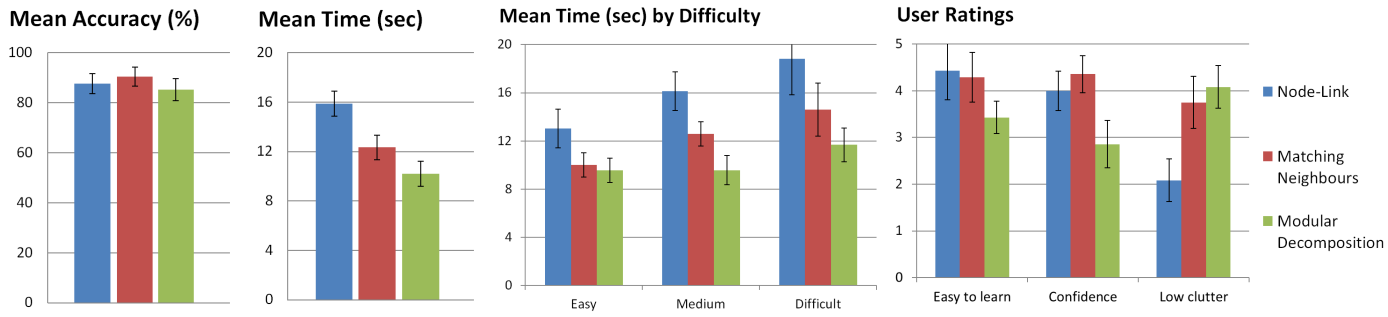


Fig. 4. Experimental results: accuracy, time, time split by dataset difficulty and user ratings.

4 Controlled Experiment 1

To assess the *readability* of diagrams produced by edge compression techniques compared to flat node-link diagrams, we performed a controlled experiment involving 15 human subjects. To evaluate the *learnability* of these techniques, we recruited participants with extremely low (or nonexistent) knowledge in graph theory. None of these participants were programmers and none used graph diagrams in their daily activities. We performed a within-subject experimental design: 3 Techniques × 3 Difficulties × 6 Repeats.

4.1 Techniques

In this initial experiment, we compared flat **node-link** diagrams (our control condition) to the **matching neighbors** and **modular decomposition** techniques described earlier. We did not include the power graph technique in this study to keep it to a reasonable time and limit our participants' fatigue. We conjectured that power graphs would require the most training and possibly feature concepts that a naïve audience may not successfully grasp. Therefore, we decided to first evaluate the simplest of our edge-compression techniques.

Layout of the graphs used in the study was accomplished by constrained-force directed layout [10]. We also made some manual fine adjustments to resolve obvious layout issues and maximize readability. It is important that all edges are at least visible and distinguishable from other edges along their entire span and that they do not cross node or module boundaries where it is not necessary to do so. The only edge routing methods that we know of that are capable of doing this are orthogonal routing, such as [25] and the "ordered bundling" edge routing method proposed by Pupyrev *et al.* [22]. Since orthogonal routing introduces right-angled bends which may make them difficult to follow, we use the technique of Pupyrev *et al.* for all the examples in this paper and for the renderings generated for the study. The DGML graphs used (viewable in Visual Studio 2012 Pro) and the images generated for the study are available from our online repository [3].

4.2 Difficulties

We generated synthetic graphs to ensure equivalent graph structures. In particular we wanted diagrams where we could control the number of nodes and the number of modules when the various techniques were applied. To achieve this we created the desired number of nodes and from this node set we randomly selected subsets of limited size to be modules. We then randomly picked pairs of modules to be linked by creating edges for the Cartesian product of their members. We applied this process recursively until the desired density or nesting level was achieved. The maximum module size and the probability of linking modules also controlled the density. The properties of each graph used are shown in Table 1. The graph *Easy3* is shown in Fig. 1.

4.3 Task

We hypothesized that, since our representation was replacing a set of links by a different shape, a connectivity task in which participants have to count or follow links would be most affected. Thus, we selected a **shortest path** task, in which participants had to count the smallest number of links to go from a node labeled "Start" to a node labeled "End". Path length varied up to 4 links. In some cases no path existed.

	Node Link		Matching Neighbors		Modular Decomposition		
Graph	$\|V\|$	$\|E\|$	m	$\|E\|$	m	$\|E\|$	nl
Easy1	7	16	1	12	2	9	1
Easy2	7	19	1	14	2	11	0
Easy3	7	23	1	18	3	9	1
Medium1	10	22	2	13	4	6	0
Medium2	10	40	3	21	5	8	1
Medium3	10	34	2	12	3	7	0
Difficult1	15	44	4	29	5	15	2
Difficult2	15	42	4	34	5	17	1
Difficult3	15	66	2	55	6	12	2

Table 1. Graphs used in study 1 and their properties for each technique, where: $|V|$ = number of nodes, $|E|$ = number of visible edges, m = number of modules, nl = nesting level.

4.4 Participants and apparatus

We recruited 15 participants (7 males and 8 females). The age of our participants ranged between 24 and 39 years (mean 31.4). The participant pool excluded programmers and participants with graph theory background. We also screened for participants with normal or corrected-to-normal vision, none being color-blind. The experimenter ran 1.5 hour sessions with two participants at a time. Each participant completed the experiment on an individual PC, using a 19-inch monitor with a resolution of 1280 by 1024.

4.5 Procedure

Training. Since our participants were not familiar with graph theory concepts, we decided to train them before each technique and consider the training successful when they could answer correctly each of the 12 training trials. The experimenter first instructed them on how to read the type of diagram using a printed version and illustrating the diagram "rules" with examples of the shortest-path task. Training both participants at the same time was beneficial as it fostered discussion and encouraged them to ask many questions. After this general training the experimenter demonstrated the experimental interface and asked participants to answer the 12 training trials individually, pausing after each trial. These trials were designed to cover all principles of readability and included task instances considered most error-prone. Overall training for the three techniques lasted up to 20 minutes. The experimenter noted that the modular decomposition technique, if taught first, led to a longer training. The most challenging concept to acquire with this technique concerned the different levels of nesting.

Experiment. We counterbalanced the order of visualization techniques across participants. We fixed the order of the datasets from low to high difficulties. For the repetitions within the same difficulty, we created isomorphic questions by either selecting different sets of sources and destinations or using a graph with similar properties. Graphs and tasks were repeated across techniques with scrambled labels. The visual representations obtained were also strikingly different as the different number of edges and the groupings led to very different layout and edge routing configurations.

The experiment software displayed the task as well as multiple choice answers and recorded accuracy and time taken for each answer. The software first displayed the question without any visualization. Once participants read the text and were ready to answer the question,

they clicked a button to view the corresponding visual. Participants pressed space bar once they found the answer. At this point, the visual was removed and the timer stopped. Participants entered their answer and reached the next question. They were instructed to rest if needed between questions and to answer questions as accurately and as fast as they could. To keep the study at a reasonable length, we limited each question to a maximum of 60 seconds. Finally, after the experiment, we collected user preferences and comments using a printed questionnaire. The study lasted approximately 60 minutes including training and post-experimental questionnaire.

4.6 Hypotheses

We formed the following hypotheses:
(H1) Since link clutter is a key factor affecting the readability of node-link diagrams, we hypothesized that modular decomposition would perform best across all difficulties, followed by matching neighbors, followed by node-link diagrams.
(H2) Modular decomposition will yield more accurate and faster results than the other two techniques for difficult datasets since these benefit most from the removal of links.
(H3) Node-link diagrams will yield more accurate results than the other two techniques in easy datasets since they are familiar to a wide audience. However, they will not yield faster results than the other two techniques for correct trials since they require following many more links than in the other two representations.
(H4) Since node-link diagrams are cluttered and modular decomposition requires learning, we hypothesized that the matching neighbors technique will be preferred by participants.

4.7 Results

We excluded one participant from our analysis since his mean accuracy (59%) was lower by more than three standard deviations compared to the mean accuracy of the remaining 14 participants (88%). However, this participant did not fail the training and his mean accuracy was much lower with node-link diagrams (39%) than with the edge compression techniques (about 60%).
Accuracy. Since accuracy does not follow a normal distribution, we used Friedman's non-parametric test. Friedman's did not reveal any overall significant difference between techniques in terms of accuracy. Participants were about 85% to 90% accurate with all techniques (Fig. 4). When splitting results by difficulty, we were surprised that Friedman's test did not reveal any significant difference between techniques either in difficult (H2) or easy (H3) datasets.
Completion time. We analyzed the completion time for correct trials with a Mixed Linear Model (MLM) capable of handling missing values. We excluded about 12% incorrect trials. Since the distribution was skewed, we used the logarithm of the completion time as is common practice. MLM revealed a significant effect of Technique ($F_{2,26} = 65.10, p < .0001$) and Difficulty ($F_{2,26} = 33.32, p < .0001$). Pairwise comparisons revealed that modular decomposition is 17% faster than matching neighbors ($p < .0001$), 36% faster than node-link ($p < .0001$) and matching neighbors is 23% faster than node-link across all difficulties (Fig. 4). These results confirm our hypothesis (H1): for correct trials, modular decomposition performs fastest, followed by matching neighbors, followed by our node-link control condition. When splitting results by difficulty, one can observe the same trend in all difficulties (Fig. 4). Pairwise comparisons revealed that node-link is significantly slower than the two other techniques in all three difficulties ($p < .001$). However, pairwise comparisons revealed that modular decomposition significantly outperforms matching neighbors for only the medium difficulty ($p < .0001$).
User Preferences. A questionnaire gathered user preferences using 5-point Likert scales. Since these results do not follow a normal distribution, we analyzed user preferences ratings using Friedman's non-parametric test. Friedman's test reveals significant difference for all questions (Fig. 4). As expected (H4), modular decomposition was found to be significantly harder to learn than the other two techniques ($p < .05$) and, despite their faster completion time, participants reported being less confident about their answers with this technique ($p < .01$). Node-link was also reported to be significantly more cluttered than the other two techniques ($p < .01$). Overall, the user ranking of the techniques revealed a significant preference of the matching neighbors technique over the modular decomposition technique ($p < .01$).

4.8 Discussion

We had hypothesized that while modular decomposition would yield more accurate results for difficult (denser) graphs, the technique might prove less effective than node-link diagrams for easy (sparser) ones. Surprisingly, our experimental results did not show any significant differences between techniques in terms of accuracy for either easy or difficult datasets. However, our results on completion time indicate that **edge compression techniques provide significant advantages over standard node-link representations**. In particular, modular decomposition outperforms the other two techniques, increasing the speed of answer by 36% on average compared to the control condition. These results clearly demonstrate the power of edge-compression techniques and suggest a clear benefit of these techniques for the representation of directed graphs.

Despite these potential benefits we had strong beliefs that the edge compression techniques (especially the modular decomposition exhibiting multiple levels of nesting) would prove difficult to learn by a naïve audience. While the difficulty of learning these techniques was reflected in the user ratings and overall ranking preference, we were extremely surprised that **all of our participants with low expertise in graph theory could successfully learn to decode these techniques in a short amount of time** (15 to 20 minutes of training overall).

5 More Sophisticated Decompositions

Thus, we see that for dense directed graphs people are able to usefully interpret aggregate edges. Furthermore, we see that the greater edge reduction provided by modular decomposition provides significant performance advantages over exact-neighbor matching or flat node-link diagrams. A natural question to ask is whether the pattern continues for power graphs, despite the additional complexity of edges crossing module boundaries. In order to answer this question we need to know how to compute the power-graph decomposition.

Computing power-graph decompositions is significantly harder than computing the graph's modular decomposition. In fact we conjecture that it is NP-hard since it appears to rely in the worst case on evaluating an exponential number of possible module choices. Unlike the modular decomposition the best power-graph decomposition is not unique. Furthermore, it is not clear what exactly we want to optimize. The obvious quantity to minimize is the number of edges in the power graph, however—as we see in Fig. 5—minimizing edges alone may lead to diagrams in which modules are highly nested and edges cross many module boundaries. We might also want to trade-off the number of modules.

As described in §3, Royer *et al.* [23] give a greedy heuristic for computing a power-graph decomposition which minimizes the number of edges. However, we do not know how effective this heuristic is or even if this is the correct feature to minimize. In order to answer these questions we used a methodology that to the best of our knowledge has not previously been used in visualization research. Our approach was to use a high-level modelling language to declaratively specify the properties we want in a good layout, including a precisely defined objective function, and then use Constraint Programming (CP) to find a provably optimal solution. The use of a high-level modelling language allowed us to quickly experiment with different aesthetic choices and the use of a provably optimal solving technique meant that differences in layout were a result of the aesthetic criteria not of vagaries of the solving heuristic.

Note that we are not (yet) advocating these optimal techniques as a practical solution for generating power graphs, especially for interactive systems. Finding the optimum takes too long and we are limited to relatively small graphs. However, it is very useful as a way to explore the parameter space for the goal function. Particularly in combination with controlled studies to evaluate the different optimal solutions, we

believe this is a powerful methodology that allows us to separate algorithm engineering from discovering the true requirements for useful visualizations. Once those requirements are understood, our intention is to design practical heuristics that give output that is closer to optimal than that of current heuristics.

5.1 Computing the Optimal Power-Graph Decomposition

In this section we describe the high-level model we used to compute the optimal power-graph decomposition and how we implemented it in MiniZinc [19]. MiniZinc is a relatively new language designed to allow high-level solver independent modelling of constrained optimization problems. Like other mathematical modelling languages, a MiniZinc model contains input parameters whose value is given in an input file, decision variables whose values are computed by an underlying constraint solver, high-level constraints and an objective function to be minimized or maximized.

The input to our problem is the number of vertices nv and a Boolean array $edge$ where $edge[u,v]$ iff there is an edge from vertex u to v. The main decision variables in the problem are the number of modules anm and the vertices in each module. We model this in our MiniZinc model by an array of Boolean decision variables where $module[v,m]$ will be set to true iff vertex v is in module m. A MiniZinc model does not specify how to compute the value of the decision variables, rather the model places constraints on the values the variables can take and it is the job of the underlying constraint solver to compute the values.

We require that the modules form a hierarchy: one way of stating this is that for all modules m and n, $m \subseteq n \vee n \subseteq m \vee m \cap n = \emptyset$. Note that the hierarchy requirement means that we can restrict $anm \leq nv$. The MiniZinc code to ensure the modules form a hierarchy is simply:

```
constraint forall(m in modules,n in modules)(mcontains[m,n]=
  forall(v in vertices)(module[v,n] -> module[v,m]));
constraint forall(m in modules, n in modules where m != n)
  (mcontains[m,n] \/ mcontains[n,m] \/
  forall(v in vertices)(not module[v,n] \/ not module[v,m]));
```

Note that the above constraints make use of the Boolean array $mcontains[m,n]$ which is constrained to hold if module m contains module n.

For any fixed choice of modules there is a unique best choice of edges in the power graph. To simplify the representation of power graph edges in the model we add a singleton "trivial" module $\{v\}$ for each vertex v:

```
constraint forall (v in vertices)(module[v,v] /\
  forall(u in vertices where u != v)(not module[u,v]));
```

This means that the edges in the power graph are all pairs of modules (m,n).

We first compute for each pair of nodes m and n if there is a *possible* edge between them. There is a possible edge between m and n iff: (1) for all $u \in m$ and for all $v \in n$, $(u,v) \in E$, and (2) $m = n$ or $m \cap n = \emptyset$. The first condition ensures the edge is semantically correct while the second condition disallows edges between ancestors or descendants in the hierarchy.

We now compute the *actual* edges in the power graph. This is any possible edge (m,n) which is not *dominated* by some other possible edge (m',n') where (m',n') dominates (m,n) if $m \subseteq m'$ and $n \subseteq n'$. The intuition is that we choose the edges as high up in the module hierarchy as possible.

The MiniZinc constraints that encode this computation are as follows. They make use of the arrays of Boolean decision variables: $pmvedge[m,v]$ iff $\exists$ a possible edge from module m to vertex v; $pvmedge[m,v]$ iff $\exists$ a possible edge from vertex v to module m; $ppgedge[m,n]$ iff $\exists$ a possible edge from module m to n; and $apgedge[m,n]$ iff $\exists$ an actual edge from module m to n:

```
constraint forall(m in modules, v in vertices)(pmvedge[m,v]=
             forall(u in vertices)(module[u,m]->edge[u,v]));
constraint forall(v in vertices, m in modules)(pvmedge[v,m]=
             forall(u in vertices)(module[u,m]->edge[v,u]));
constraint forall(m,n in modules)(ppgedge[m,n] =
             (forall(v in vertices)
               (module[v,n] -> pmvedge[m,v]) /\
               (m=n \/ forall(v in vertices)
                 (not module[v,n] \/ not module[v,m])) ));
constraint forall(m,n in modules)(
            apgedge[m,n] = (
              ppgedge[m,n] /\ m <= anm /\ n <= anm /\
              forall(p in nv+1..nm)(
                (mcontains[p,m] -> not ppgedge[p,n]) /\
                (mcontains[p,n] -> not ppgedge[m,p])) ));
```

The final piece of the puzzle is the objective function which we wish to minimize. This is specified in our MiniZinc model by

```
solve minimize numbermodules + edgeweight*numberedges +
               crossingweight*numbercrossings;
```

This states that the objective function is the weighted linear sum of the number of modules, number of power graph edges, and the number of edge/module boundary crossings. The weights are input parameters: by changing these in the input file it is simple to explore the layout tradeoff between these features. It is also simple to add other features that we wish to explore to the objective function.

We must also add constraints to ensure that the decision variables *numberedges*, *numbermodules* and *numbercrossings* are correctly computed. The following MiniZinc assignment statements do this for edges and modules:

```
numbermodules=anm-nv;
numberedges=sum(m,n in modules)(bool2int(apgedge[m,n]));
```

The number of edge crossings is a little trickier: to help we compute the array of integer decision variables, $mcrossings[m,n]$, which gives the number of module boundaries that must be crossed to get from module m to module n. This is simply the number of modules p that contain either m or n but not both.

```
constraint forall(m,n in modules)(
            mcrossings[m,n] = sum(p in modules)(
              bool2int(mcontains[p,m] xor mcontains[p,n])));
numbercrossings=sum(m,n in modules)(
                  bool2int(apgedge[m,n]) * mcrossings[m,n]);
```

Running Time As one might expect this model is quite slow even with the *cpx* solver (which is one of the fastest solvers supporting MiniZinc). We improved efficiency by adding a number of redundant constraints and constraints to remove symmetric solutions. While the model is still too slow for real-world layout it can compute optimal power-graph decompositions for graphs with 10-15 nodes, allowing us to determine a ground truth corpus for evaluating heuristic techniques and also to generate layouts for exploring the tradeoffs between different layout features.

Interestingly, the running time is not necessarily strongly tied to the number of nodes or edges, rather it is related to the number of modules in the optimal solution. Instances with only a couple of modules in the optimal solution are generally solvable in several minutes on a standard 2013 PC[1] while, for example, Difficult3 Non-Modular with 7 modules took over 8 hours. Note also, that often the optimal solution is found relatively quickly (e.g. in the first half of the total running time) but significantly more time is required to prove optimality.

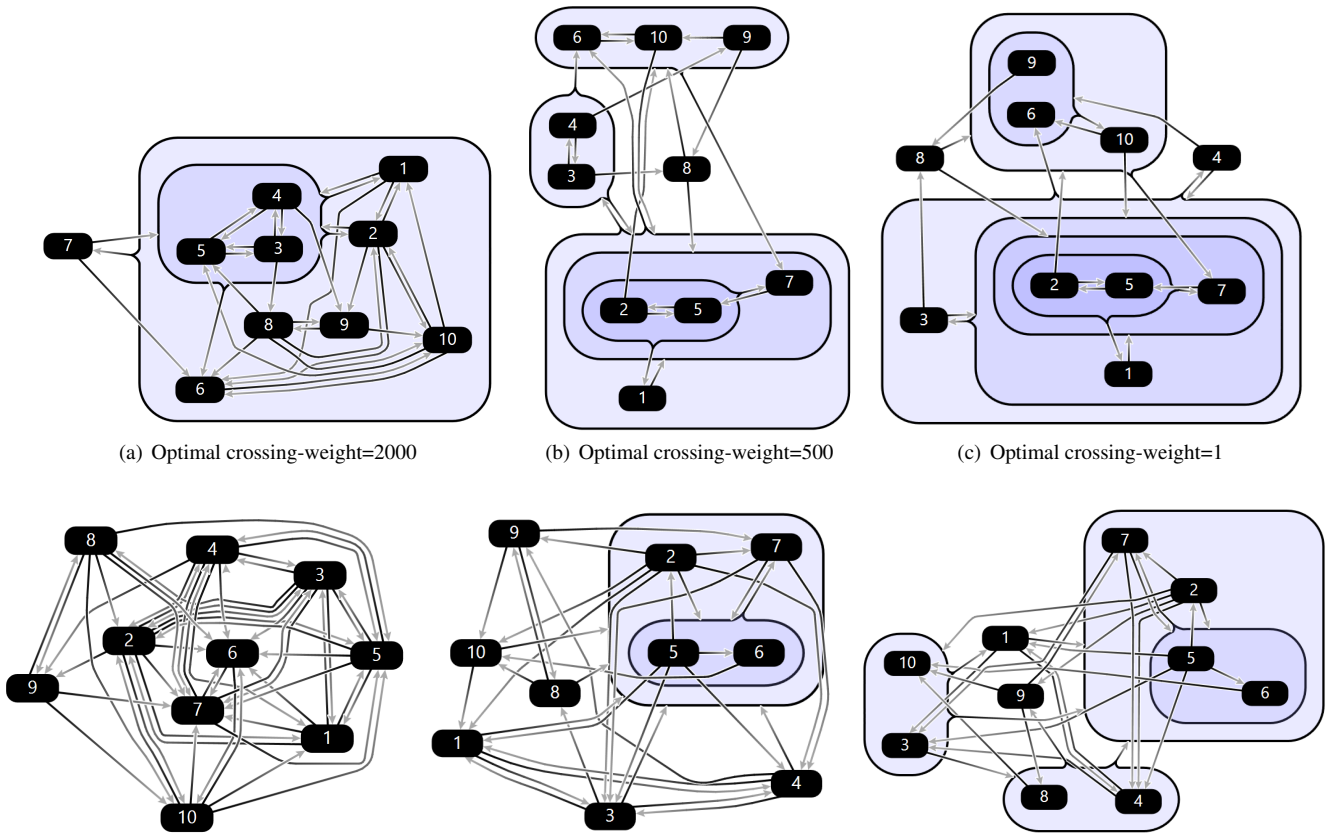
By contrast the heuristic described in §3.3 is $O(|V|^2 \log |E|)$ with efficient set comparison techniques and runs in under a second on all of our test graphs. The decomposition of a graph with $|V| = 140$ and $|E| = 406$ shown in Fig. 9 took 39 seconds on the same hardware. An optimal compression of such a large graph is not practical with current technology.

6 Compressed Graph Corpus

To evaluate the amount of compression gained by the optimal decompositions over the power-graph heuristic we prepared a corpus of 28 dense graphs.

We prepared eight graphs for our second study, based on the medium and difficult graphs used in the first study. The idea of reusing

[1] Intel Ivy Bridge Core i7, up to 2.6GHz



(a) Optimal crossing-weight=2000 (b) Optimal crossing-weight=500 (c) Optimal crossing-weight=1

(d) Heuristic crossing-weight=2000 (e) Heuristic crossing-weight=500 (f) Heuristic crossing-weight=1

Fig. 5. A small but dense scale-free graph and the results of optimal and heuristic power-graph decomposition for various crossing-penalties. In all cases the edge-weight was kept fixed at 1000.

graphs is to provide continuity between studies. Two of them (*Difficult Modular*) are unmodified from the first study, and so permit a modular decomposition. The rest have been made more dense—and so more interesting for power graph analysis—by randomly adding edges until a modular decomposition is no longer possible. They have 10 (*Medium Non-Modular*) or 15 (*Difficult Non-Modular*) nodes and between 24 and 70 edges.

In addition we generated a set of twenty random scale-free graphs. These were generated following the model of Bollobás *et al.* [7] which has been shown to construct organic looking graphs with similar structure to those found in biological systems and complex evolving artifacts like the web. We modified the model to eliminate self-loops and multiple edges by simply discarding any such edges. Also, we limited the size of the generated graph to n nodes by stopping the algorithm immediately before the $(n+1)^{\text{th}}$ node would be added.

The Bollobás *et al.* model has five parameters. At each discrete time step, the graph grows in one of three ways: (a) a new node with a single outgoing edge is added, (b) a new edge between existing nodes is added, or (c) a new node with a single incoming edge is added. The parameters α, β and γ are the respective probabilities of each possibility. When choosing a destination for the edge in (a) or (b), the choice is weighted towards nodes that have a high in-degree, and when choosing an origin for the edge in (b) or (c), the choice is weighted towards nodes that have a high out-degree. The weighting is influenced by parameters δ_{in} and δ_{out}, where higher values of the parameters give less weight to the existing in- and out-degrees.

To generate our corpus of graphs, we used a node limit of 10. After some experimentation we found that the parameters $\alpha = \gamma = \frac{1}{42}$, $\beta = \frac{40}{42}$, and $\delta_{in} = \delta_{out} = 1$ generated graphs with a good distribution of edge densities, between 26 and 60 edges.

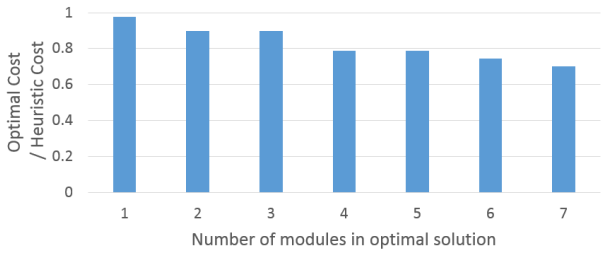


Fig. 6. The optimal/heuristic cost ratio broken down by number of modules in the optimal solution.

7 Effectiveness of Compression

We then ran our MiniZinc solver on each of these 28 graphs, with three different settings for the crossing-weight parameter in the objective function, to find a total of 84 graph decompositions that are optimal with respect to the given goal function. This graph corpus and their optimal decompositions are available online [3] and constitute a first benchmark useful for further evaluations by the research community.

The crossing weights used were 2000, 500 and 1 while the edge weight was kept fixed at 1000. We obtained 84 more decompositions using the greedy power-graph heuristic to obtain approximate solutions, also varying the crossing-weight used to decide when to create a module in the heuristic's cost function. An example set of results for one graph is given in Fig. 5. We found that with crossing weight of 2000 the optimal solution was exactly the modular decomposition on our two modular graphs. On the denser modular graph the heuristic failed to find one of the modules.

It is interesting to compare the cost as computed by the objective function for the optimal versus approximate solutions. In Fig. 6 we compare the ratio of optimal cost to heuristic cost against the number of modules in the optimal solution. The heuristic does well with very few modules but deteriorates as this number increases. A ratio of 0.7 means a difference of dozens of edges or crossings depending on

the objective function. We see in Fig. 5 that this makes a significant difference.

8 Controlled Experiment 2

Our first experiment revealed that participants could successfully *learn* a complex edge-compression technique involving several levels of nesting and *significantly increase their performances* for shortest path tasks. Yet, as described in §5 we can go further by allowing edges to cross module boundaries. Our next study investigates how different degrees of compression affect readability.

This second controlled experiment involved 14 human subjects, again with extremely low or no knowledge in graph theory. None of these participants were involved in the first study and none used graph diagrams in their daily activities. We used a within-subject experimental design: 3 Edge compression levels $\times$ 3 Difficulties $\times$ 6 Repeats.

8.1 Degree of edge crossing

We believe that the number of edges crossing module boundaries is the factor that most impacts the readability of power graphs. Therefore, we selected three conditions with **low**, **medium** and **high** numbers of cross-boundary edges. These correspond to the optimal solutions obtained with *crossingweight* $= 2000, 500, 1$ respectively, used in the objective function of our power graph model as described in §7.

8.2 Difficulties and Task

We chose graphs intended to provide three difficulty levels: **Difficult Modular** were the unmodified difficult graphs from Study 1; **Medium Non-Modular** and **Difficult Non-Modular** were modified from Study 1 as described in §6 so that they no longer afford a modular decomposition. That is, it is no longer possible to compress these latter graphs without creating crossings between edges and module boundaries. Medium Non-Modular graphs look similar to Fig. 5. We selected the same **shortest path** task as in study 1.

8.3 Participants and apparatus

We recruited 14 participants (7 males and 7 females). The age of our participants ranged between 20 and 38 years (mean 29.4). The participant pool excluded participants of the first study, programmers and participants with graph theory background. We also screened for participants with normal or corrected-to-normal vision, none being colorblind. The experimenter ran 1.5 hour sessions with two participants at a time using the same apparatus as experiment 1.

Layout was as described in the first study although the visuals became more polished, following recommendations for directed edge representations from Holten *et al.* [14]. Again all materials are available [3].

8.4 Procedure

Training. Since our participants used the same technique (with different levels of crossings), we only trained them at the beginning of the experiment. The experimenter followed the same training protocol as in the first study. Overall training for the technique lasted 15 minutes on average. The experimenter noted that the most challenging concepts to acquire were the different levels of nesting and edges crossing boundaries.

Experiment. We counterbalanced the order of the degree of crossing across participants. We fixed the order of the datasets from low to high difficulties. For the repetitions within the same difficulty, we created isomorphic questions by either selecting different sets of sources and destinations or using a graph with similar properties. Each level of compression displayed the same exact graphs and tasks. Similar to the first study, the visual representations obtained had very different layouts and edge routing configurations and we randomized labels in all diagrams to avoid memorization.

We used the same experimental software as in the first study. After the experiment, the experimenter briefly interviewed participants on what they found most difficult or confusing in these diagrams. The study lasted approximately 60 minutes including training and post-experimental interview.

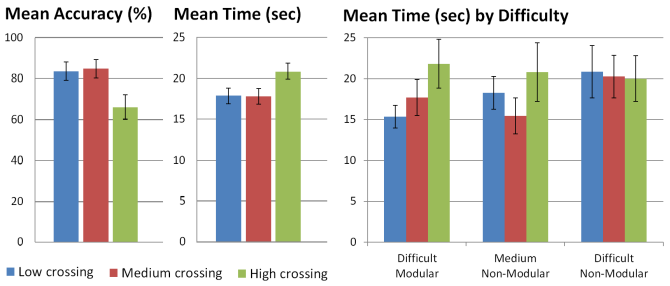


Fig. 7. Experimental results: accuracy, time and time split by dataset difficulty.

8.5 Hypotheses

We hypothesized (H5) that a medium degree of cross-boundary edges—as a compromise between compression and complexity—would give best performance. We further hypothesized (H6) that the highest degree of cross-boundary edges would prove too complex to read for a naive audience, thus yielding less accurate and slower results.

8.6 Results

Accuracy. Since the accuracy does not follow a normal distribution, we used Friedman's non-parametric test. Friedman's test reveals an overall significant difference in accuracy between techniques ($p < .0001$). Pairwise comparisons using the Wilcoxon's test showed that the high level of cross-boundary edges leads to more errors than the other two conditions ($p < .0001$), verifying (H6). Participants were only 66% accurate in this condition contrasting with the 84% and 85% mean accuracies for the low and medium crossing levels (Fig. 7). When splitting results by difficulty, Friedman's and Wilcoxon's tests reveal the same trend for Difficult Modular ($p < .0001$) and Medium Non-Modular ($p < .0001$) graphs: a high degree of crossings yields more errors than the other two conditions. However, there is no significant difference between crossing degree for the most difficult case.

Completion time. We analyzed the completion time for correct trials with a Mixed Linear Model (MLM) capable of handling missing values. We excluded 22% incorrect trials. Since the distribution was skewed, we used the logarithm of the completion time as is common practice. MLM revealed a significant effect of degree of crossing ($F_{2,26} = 5.77, p < .01$) and Difficulty ($F_{2,26} = 3.61, p < .05$). Pairwise comparisons revealed that overall, the medium degree of crossing outperforms the high one ($p < .0001$) (Fig. 7).

When splitting by difficulty, one can observe different trends for each type of dataset (Fig. 7). It is interesting to note that in the modular datasets, a low degree of crossing equates to the modular decomposition technique. Our results indicate that participants can handle a moderate number of edges crossing module boundaries (H5) as low and medium level do not yield significantly different times but both prove significantly faster than the high degree of crossing ($p < .05$). For the non-modular datasets, the medium degree of crossing leads to faster completion time than the other two techniques for the medium graphs ($p < .01$) but there is no significant difference for the difficult graphs. These results may imply that as the graph gets denser (and thus requires more time for finding a shortest path between two nodes), the degree of crossing does not have a noticeable impact on the completion time anymore.

User comments. After the timed trials, the experimenter collected comments about the difficult or confusing aspects of these diagrams. The experimenter did not prompt participants with particular aspects but rather collected spontaneous reactions of the participants. We categorized the difficult aspects described by our participants into two major areas: cross-boundary edges and nesting. Nine out of 14 participants commented that the cross-boundary edges (especially from a node inside a module to a different module) were the most difficult aspect of these diagrams. Five participants explicitly said that these were the most likely cause of their errors. Four of our 14 participants identified nesting as causing them problems; three of these did not mention

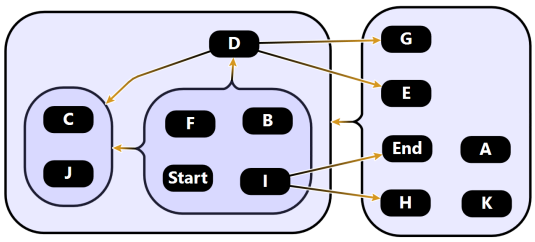
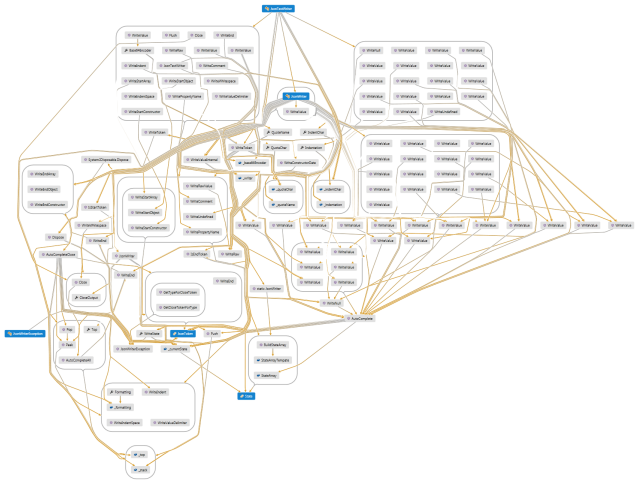


Fig. 8. A fiendishly tricky task used in training for the second study. The shortest paths are 4 steps: e.g. Start→D→G→I→End.

(a) Flat graph with 140 nodes and 406 links.



(b) A compressed version (using the heuristic described in §3.3) has 17 modules, reducing the visible edge count to 208. Obviously, at this scale individual edges are difficult to follow, yet the groupings are also interesting since they show the members that are accessed together.

Fig. 9. A mapping of the reference graph for five classes (blue nodes) and their members from the Json.NET library [2].

the cross-boundary edges at all. Finally, one participant commented that, at first, the most confusing aspect of these diagrams was the affordance of the visual grouping into modules. He explained that such grouping naturally conveyed to him that nodes within a module are directly connected (clique). While training helped, he struggled against this instinctive decoding of the diagram throughout the trials.

9 Discussion

Our first study revealed that edge compression through modules does significantly improve response times in path following tasks once the reader has understood the concept of inferred edges. While this concept is not immediately intuited by most novices, 10–15 minutes of training seems adequate to make most of them proficient. Our second study showed that allowing some cross-module edges (and hence allowing compression of graphs that do not permit a modular decomposition) is still readable though compression methods should apply a penalty to cross-edges. Our medium crossing condition ($edgeweight = 1000$, $crossingweight = 500$) seems to offer the best compromise in most graphs.

To qualify, however, our results regarding cross-edges did vary in different graphs. To demonstrate why certain graphs and path following tasks can be particularly hard we give a final small example. Fig. 8 is a task used in training that requires both following cross-module edges and unrolling nested modules.

Scope and Limitations. This paper focuses on directed graphs instead of general undirected graphs for several reasons:
- There are many applications that interest us where the directionality of edges is very important.
- As already mentioned, the problem of edge density is potentially much greater for directed graphs.
- The Power Graph Analysis technique has not previously been considered for directed graphs.
- Greater care must be taken with the visual representation to ensure edge direction is clearly readable.

We limit the evaluations in this paper to graphs with relatively few nodes, again for multiple reasons:
- In our trial runs for the first study we began by considering larger graphs (up to 30 nodes), however the participants were overwhelmed and the flat graphs were completely unreadable. Even at 13 nodes, Fig. 2(a) is becoming overwhelming.
- There is no doubt that practical applications require analysis of large graphs, yet small graph visualization is still applicable for detailed examination of a neighborhood or viewing at a high level of abstraction. For example, the graph in Fig. 2 is only the top-level semantic grouping of a graph with hundreds of thousands of nodes.
- For smaller graphs it is possible to determine optimal decompositions which affords precise evaluation as described above.
- Our feeling after completing the studies, is that the difficulty in performing path following tasks has less to do with the number of nodes but everything to do with the complexity of path in terms of whether it crosses module boundaries or requires the reader to mentally unroll nested modules, §8.

Still, scalability to larger instances is essential future work. In Fig. 9 we show heuristic compression of a graph with 140 nodes and 406 edges to demonstrate that—even though individual edges are no longer easily traceable without interaction—compression still results in a significantly less cluttered visualization and there is utility in simply obtaining layout that respects the modules. In this software dependency graph, the modules themselves are also meaningful in that they group class members that are accessed together. Further refinement of the algorithmics, visual design and interaction to such larger examples is interesting, but beyond the scope of this paper.

Future Work. There are many directions for future work. We are developing a Visual Studio extension for producing "Poster Views" of code dependencies as in Fig. 2(c). This extension and the source code for all the techniques described are available [1]. We make our corpus of test graphs and optimal power-graph solutions available in the same repository as detailed in [3]. These optimal solutions represent hundreds of hours of processing time and should be a valuable resource for both those interested in performing experiments on different representations and algorithms engineers interested in developing more optimal heuristics. For those interested in experimenting with different goal functions or different input graphs our MiniZinc model is also available.

We find this methodology of using constraint programming to declaratively model and optimally solve small instances of difficult graph visualization problems compelling, as it avoids the shortcomings of heuristics in the exploration of the design space. We hope to investigate this approach further in different areas such as layout.

Regarding power graphs generally, there is much work to be done: as demonstrated in §7 we need more optimal heuristics; we need stable techniques for dynamic graphs; we need to extend to graphs with multiple edge types and attributed edges. Another popular technique for removing clutter in dense graphs is edge bundling, e.g. [15]. However, usually groups of edges are bundled based on spatial locality. When the edges in such a bundling run concurrently this is effectively a lossy compression as exact connectivity is no longer discernible. An obvious alternative which would not result in this ambiguity is to bundle edges based on the decompositions described in this paper.

REFERENCES

[1] DGML PosterView project home page on codeplex: https://dgmlposterview.codeplex.com, Accessed June 2013.

[2] Json.NET home page http://json.codeplex.com, Accessed June 2013.

[3] Wiki page with links to our various study materials and optimal compression solution corpus: https://dgmlposterview.codeplex.com/wikipage?title=Study, Accessed June 2013.

[4] IronPython home page http://ironpython.net, Accessed March 2013.

[5] J. Abello, F. van Ham, and N. Krishnan. Ask-graphview: A large scale graph visualization system. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):669–676, 2006.

[6] D. Archambault, H. Purchase, and B. Pinaud. The readability of path-preserving clusterings of graphs. *Computer Graphics Forum*, 29(3):1173–1182, 2010.

[7] B. Bollobás, C. Borgs, J. Chayes, and O. Riordan. Directed scale-free graphs. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 132–139. Society for Industrial and Applied Mathematics, 2003.

[8] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. On finding graph clusterings with maximum modularity. In *Graph-Theoretic Concepts in Computer Science*, pages 121–132. Springer, 2007.

[9] K. Dinkla, M. A. Westenberg, and J. J. van Wijk. Compressed adjacency matrices: Untangling gene regulatory networks. *IEEE Transactions on Visualization and Computer Graphics*, 11(12):2457–2466, 2012.

[10] T. Dwyer and G. Robertson. Layout with circular and other non-linear constraints using procrustes projection. In *Proceedings of 18th International Symposium on Graph Drawing (GD'10)*, volume 5849 of *LNCS*, pages 393–404. Springer, 2010.

[11] A. Ehrenfeucht, H. N. Gabow, R. M. McConnell, and S. J. Sullivan. An $O(n^2)$ divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs. *Algorithms*, 16:283–294, 1994.

[12] E. R. Gansner, Y. Koren, and S. C. North. Topological fisheye views for visualizing large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):457–468, 2005.

[13] J. Heikniemi. What's new in .NET framework 4.5? [poster] www.heikniemi.net/hardcoded/2011/10/whats-new-in-net-framework-4-5-poster/, Accessed March 2013.

[14] D. Holten, P. Isenberg, J. J. van Wijk, and J. Fekete. An extended evaluation of the readability of tapered, animated, and textured directed-edge representations in node-link graphs. In *Proceedings of the Pacific Visualization Symposium (PacificVis)*, pages 195–202. IEEE, 2011.

[15] D. Holten and J. J. van Wijk. Force-directed edge bundling for graph visualization. In *Computer Graphics Forum*, volume 28, pages 983–990. Wiley Online Library, 2009.

[16] L. O. James, R. G. Stanton, and D. D. Cowan. Graph decomposition for undirected graphs. In *Proceedings of the Third Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 281–290, 1972.

[17] R. M. McConnell and F. de Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Applied Mathematics*, 145(2):198–209, 2005.

[18] C. L. McCreary, R. O. Chapman, and F. S. Shieh. Using graph parsing for automatic graph drawing. *IEEE Transaction son Systems, Man, and Cybernetics*, 28(5):545–561, 1998.

[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543. Springer, 2007.

[20] M. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[21] C. Papadopoulos and C. Voglis. Drawing graphs using modular decomposition. In *Proceedings of the 13th international conference on Graph Drawing (GD'05)*, volume 3843 of *LNCS*, pages 343–354. Springer, 2006.

[22] S. Pupyrev, L. Nachmanson, S. Bereg, and A. E. Holroyd. Edge routing with ordered bundles. In *Proceedings of 19th International Symposium on Graph Drawing (GD'11)*, volume 7034 of *LNCS*, pages 136–147. Springer, 2011.

[23] L. Royer, M. Reimann, B. Andreopoulos, and M. Schroeder. Unraveling protein networks with power graph analysis. *PLoS computational biology*, 4(7):e1000108, 2008.

[24] F. van Ham, M. Wattenberg, and F. B. Viegas. Mapping text with phrase nets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1169–1176, Nov. 2009.

[25] M. Wybrow, K. Marriott, and P. J. Stuckey. Orthogonal connector routing. In *Proceedings of 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 219–231. Springer, 2010.