UCoD - Simplifying Supply Chain Structures in the Browser

Alex Trostanovsky and Nikola Cucuk

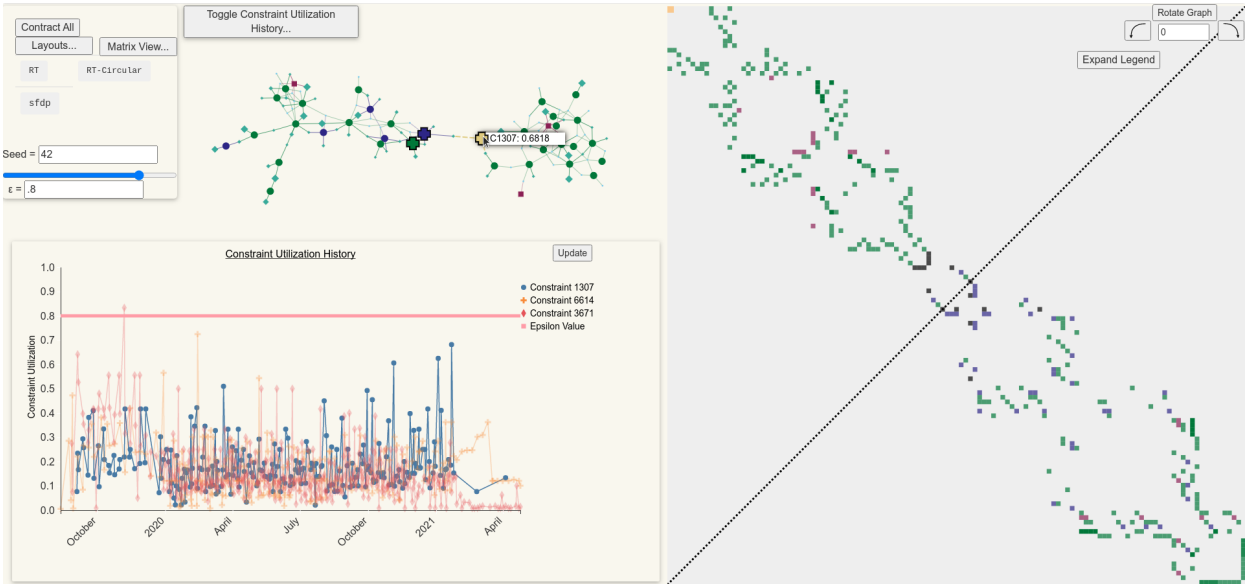


Fig. 1: The Underutilized Constraint Detector.

Abstract— Visualizing supply chain networks is difficult but important. Analyzing supply chains as node-link graphs can reveal topological information about the dependencies that exist between parts and the resources those parts depend on. In previous work, we developed a heuristic algorithm that detects underutilized components in a supply chain. In this work, we present UCoD: the Underutilized COnstraint Detector, a tool that highlights the output of this heuristic algorithm, and allows supply chain planners to visualize the structure of their supply chain, and help them identify underutilization and anomalous patterns and trends in their network. UCoD consists of a juxtaposed view of a node-link diagram and an adjacency matrix of the supply chain graph, and a superimposed line chart that allows supply chain planners to inspect time-series of utilization for the nodes in their network. This functionality allows supply chain planners to verify the output of our heuristic algorithm and simplify their product structures.

Index Terms—Supply chain management, Supply chain visualization

1 Introduction

Kinaxis is a supply chain management company that models the product structures of its customers using graphs. A supply chain product structure is a complete description of how something gets made. To schedule a product structure, one must consider all of its constituent parts. In such a calculation, certain parts may depend on others. For example, the assembly of a bicycle depends on the assembly of a cassette, which itself depends on the assembly of individual cassette-cogs. These *one-way* dependencies can be calculated by maintaining that every dependant part relies on the scheduling of its parent parts.

However, applying the same logic without additional consideration to *multi-way* dependencies could cause deadlocks and conflicting schedules. In a *multi-way* dependency, multiple parts rely on a shared constraint. For example, a pair of different-sized cogs that get assembled on the same production line. If these parts were scheduled in isolation, the resulting production schedules could conflict with one another. To address this, parts that share common constraints are grouped into structures known as **Calculation Families**.

Inclusion in calculation families is a transitive operation that can result in the merging of families. For example, if two calculation families contain parts that share common constraints, then the two families will be merged into a single calculation family (See Fig. 2). The scheduling

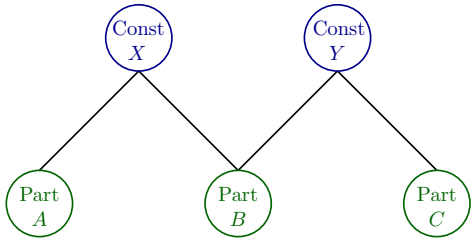


Fig. 2: Merging of Calculation Families. Since parts A and B share constraint X, they must be in the same calculation family. Similarly, Parts B and C share constraint Y, so they must be in the same family. Therefore, Parts A, B, and C are grouped into the same calculation family.

algorithms used by Kinaxis to schedule their customers' product structures are parallelized across calculation families. Therefore, the fewer families that a customer has in their product structure(s), the slower

• *Alex Trostanovsky and Nikola Cucuk are with the University of British Columbia. E-mail: atrostan@cs.ubc.ca – nca3@sfu.ca.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

the algorithms will be executed. Furthermore, having fewer families usually means that each family will be larger, further hindering parallel execution.

This project builds upon work completed in 2020 in collaboration with Carleton University's Computational Geometry Lab and Kinaxis with funding provided by the NSERC-Engage Alliance Grant. In said work, we defined a graphical model of calculation families and automated the detection of underutilized constraints in calculation families using a recursive minimum-cut removal [23] algorithm.
The mincut removal algorithm can efficiently detect underutilized constraints in calculation families and these results can be used by supply chain planners to optimize scheduling. However, before committing to the removal of underutilized constraints from a calculation family, there would be an added advantage to the visualization of the structure of the families. Since calculation families may contain 1 000s of nodes and 10 000s of edges, developing a simplified visual representation that summarizes the calculation family graph and highlights the presence of underutilized constraints would help facilitate the interaction of supply chain planners with these complex data structures.

We propose **UCoD** (The Underutilized Constraint Detector) - a tool that will allow supply chain planners to visualize, query, and search the calculation families in their product structures. This graphical representation of calculation families will take the form of an interactive, web-based, node-link diagram that will display the high-level structure of calculation families and the properties of underutilized constraints. Given this information, the user may decide to remove the highlighted constraints from the graph. The removal of these constraints from the graph will split up calculation families substantially, while still maintaining a valid and feasible scheduling plan. As a result, the parallel scheduling of the (now smaller) families will be faster.

2 Related Work

2.1 Supply Chain Management

Visualizing supply chains helps the user to:

1. Simplify the supply chain network [14],
2. Identify transportation bottlenecks or geographic concentrations [2]
3. Find alternate supply chain structures [2]

Traditionally, supply chains are represented as directed graphs or "netchains" [16]. These methods aid in the visualizations of the topology and dependencies that exist in supply chains. This visual analysis reduces the user's cognitive load and expedites exploration by projecting emergent relationships between entities [7].
However, representing supply chains using visual interfaces is still in a nascent stage. Minegishi and Thiel [19] used causal loop diagrams to represent supply chain interactions. Hu et al. [10] developed a framework for visually representing the geographical attributes of a supply chain using a case study from the transport container industry. Finally, Kassem et al. [15] developed a visualization scheme that maps relevant information to the progress of building construction (including the construction materials' supply chain).

2.2 Graph Visualization

In this work, we'll represent **Node-Link (NL)** diagrams in 2D space. These diagrams work well with small, sparse networks, but are poorly readable with large and highly connected graphs. Plotting large networks becomes difficult when nodes are placed in a constrained 2D space. A reoccurring problem that is observed when plotting large networks is the production of "hairballs" [20] - described as a visual clutter of occluded nodes and edges in large, dense graphs. To address this, node placement could be done using force-directed layouts [9, 12]. By assigning repelling forces to edges and nodes based on their relative position, these algorithms attempt to maintain that all edges are of more or less equal length and that there are as few crossing edges as possible.

Force-directed layout algorithms have difficulty converging to a placement solution in a reasonable run-time. Also, because the placement of nodes is not deterministic, identical layout reproduction can be challenging. Force-directed placement algorithms search in a way that can get stuck in local minima that may not be the optimal solution to the placement problem. To address these issues, multilevel force-directed placement algorithms have been developed [9]. These algorithms augment the original network with a derived cluster hierarchy to form a compound network. The cluster hierarchy is computed by coarsening the original network into successively simpler networks. This approach is better at avoiding local minima and can provide reproducible general placements. Finally, the run-time of multilevel force-directed placements is also improved because individual clusters are redefined independently as smaller networks.

The **Adjacency Matrix (AM)** is another way to visualize graphs. The rows and columns of this matrix are indexed by the nodes in the graph. If an edge e_{ij} exists between nodes $v_i, v_j \in G(V,E)$ (where $G(V,E)$ is the graph, and V,E are the sets of nodes and edges in G), then the i,j^{th} entry in the AM will be filled. These entries can be filled with a boolean value, an edge weight, or a colour to encode an edge attribute [17]. Such an implementation can achieve high information density, up to a limit of one thousand nodes and one million edges. An aggregated multilevel matrix view could handle up to ten billion edges [20]. While an NL diagram requires the entirety of the graph to be shown, only half of the AM needs to be shown for an undirected graph, because a link from node v_i to node v_j implies a link from v_j to v_i [20]. Matrix views don't suffer from non-deterministic placement, because the area and the dimensions of the matrix are fixed. One major weakness of matrix views is unfamiliarity: most users can easily interpret NL views but not matrix views [20].

We would like to enable supply chain planners to view the topology of their product structures after the removal of underutilized constraints. To address node and edge occlusion, we hypothesize that contracting nodes and edges into "meta-nodes" and "meta-edges" will help declutter our visualization. PivotGraph [28] combines nodes based on their similarity and displays the graph nodes on a grid with two axis-aligned, categorical attributes. Once the nodes have been contracted, PivotGraph assigns each dimension a row/column and places a vertex in the corresponding row/column for each attribute. It then draws in the connections between each vertex, with the width of the edge representing the number or strength of the connection between the vertices. Honeycomb [27] also aggregates nodes but uses a predefined hierarchy to aggregate the cells of an adjacency matrix to display social networks with millions of connections.

To view a supply chain graph, we will incorporate the NL and AM views of graphs. The NL view will allow the user to view the topology of the supply chain graph and the AM view will allow the user to intuitively assess the sizes of the two disconnected components produced by the removal of underutilized constraints.

3 Dataset and Tasks

3.1 Dataset

Kinaxis has provided the authors with an anonymized customer's dataset. Utilization of the Family Separation Dataset was conditioned upon compliance to a Non-Disclosure Agreement between Kinaxis and the authors.

Kinaxis' Family Separation Dataset contains 199 807 parts, 8 251 constraints, and 2 989 families. The dataset consists of three tables:

1. `Edges - Part to Constraint`: defines which Parts depend on which Constraints. Part and Constraint nodes are implied by this table.
2. `Edges - Part to Family`: defines which Parts belong to which Calculation Families; a key-value map between Families and their constituent parts. The inclusion of constraints in families is implied by this table. Specifically, if Family f_1 contains Part p_1, and Part p_1 is dependent on Constraint c_1, then Family f_1 contains Part p_1 and Constraint c_1.

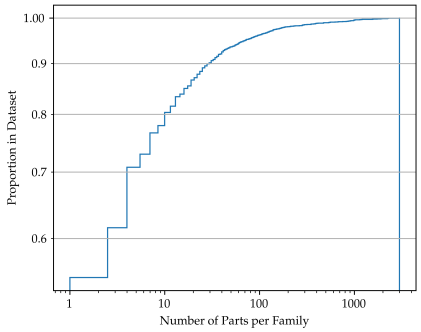


Fig. 3: Cumulative Distribution Plot: Number of Parts Per Family

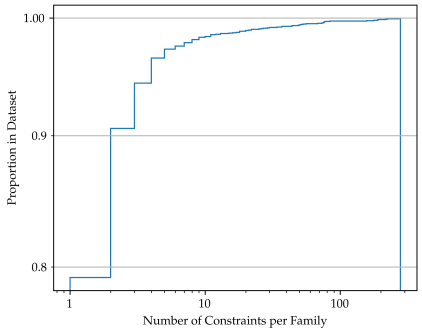


Fig. 4: Cumulative Distribution Plot: Number of Constraints Per Family

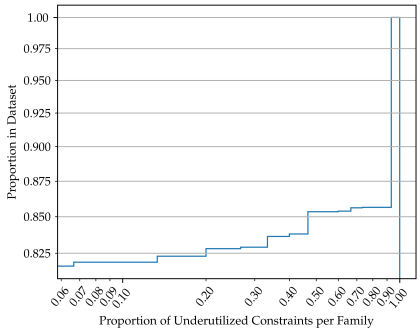


Fig. 5: Cumulative Distribution Plot: Proportion of Underutilized Constraints Per Family

3. `Constraint Utilization`: contains constraint utilization observations over time.

Every constraint utilization observation is attributed with the following features:

- The constraint *capacity*, $p \geq 0$: A scalar value that indicates the constraint's total capacity.
- The constraint *load*, $l \geq 0$: A scalar value that indicates how much of the constraint's capacity was utilized (in aggregate) by all of its dependant parts.

Using these attributes, the following metrics are derived:

- The constraint *utilization*, $u := l/p$.
- Due to the dynamic nature of constraint load and capacity, an important metric to observe is the *maximum constraint utilization*. This metric is defined to be the largest constraint utilization observed over the entirety of the dataset.
- A user-defined threshold, $0 < \varepsilon < 1$, defines constraint underutilization. If a constraint's max constraint utilization is below ε, then that constraint is **underutilized** (See Fig. 6).
- The **candidate constraints for removal** are underutilized constraint nodes in a family graph whose removal from the graph will result in two disconnected subgraphs of roughly equal size.

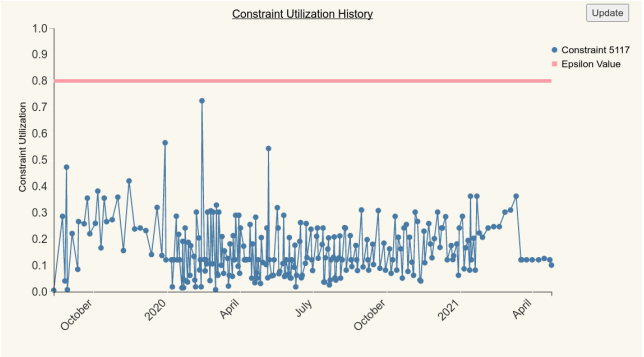


Fig. 6: The constraint utilization time-series view in UCoD. Because the constraint's utilization never exceeds $\varepsilon = 0.8$, it is considered to be underutilized.

3.2 Data Abstraction

The dataset and derived quantities and categorizations from Section 3.1 define an undirected node-link graph. Nodes in the graph are either Parts or Constraints, where a Part node is connected to at least one Constraint. The undirected edge between a Part and a Constraint indicates a *Part-Constraint Dependency*: all part nodes adjacent to a constraint node are dependent on that constraint.

There are 4 categories of constraint nodes:

1. *Underutilized Constraints*.
2. *Constraints with No Utilization History*: these are constraint nodes that belong to the calculation family, but do not have any constraint utilization observations.
3. *Candidate Constraints for Removal*.
4. *Utilized Constraints*: constraints that do not belong to any of the above categories.

Finally, constraint utilization for each constraint node is a derived attribute that defines a time-series of utilization observations for that constraint.

3.3 Task Abstraction

The goal of this work is to propose the removal of constraint nodes that would both significantly reduce the family size, and continue to represent a feasible and reasonable scheduling plan. The set of candidate constraints for removal will be identified using the recursive minimum-cut removal algorithm developed in our previous work. These candidates will be displayed to the user, who may then decide to remove the proposed constraints from the family graph. We intend UCoD to allow supply chain planners to:

1. Explore and analyze the topology of the calculation family graph.
2. Given a candidate constraint for removal, compare the sizes of the two subgraphs induced by its removal.
3. Identify anomalous constraints in the graph (constraints that were entered erroneously into the graph, or constraints that are no longer being utilized).
4. Compare utilization patterns between constraints.

Given the ability to perform these tasks, we envision two outcomes that could result from a supply chain planner's interaction with UCoD.

1. Upon inspection of the Candidate Constraint for Removal, the user may decide that the constraint *should not* be removed from the graph. This may be due to a suboptimal split of unequal sizes, or from an observed trend of increased utilization by the candidate constraint.

2. Otherwise, the user may choose to remove the candidate constraint from the graph. The family size will be substantially reduced, and the scheduling of the partitioned family will be optimized. The new scheduling calculation induced by the split-up family will be monitored by Kinaxis' supply chain planners to verify that it continues to represent a valid plan.

4 UCoD

4.1 Visual Encoding and Idiom

We display Calculation Families as Node-Link Diagrams that can be contracted and laid-out to reduce node and edge occlusion. We juxtapose this display with an adjacency matrix view. Lastly, we allow the user to select constraints in the graph to view their utilization histories.

4.1.1 Family Selector

The Candidate Constraints for Removal are displayed in a table. Each row in the table corresponds to a calculation family that has been identified by the recursive mincut algorithm to contain a candidate constraint for removal. The user can select each calculation family to display the node-link view corresponding to it.

#	Product Structure ID	Component ID
0	12197	1
1	12197	2
2	315925	2
3	396955	0
4	306080	0

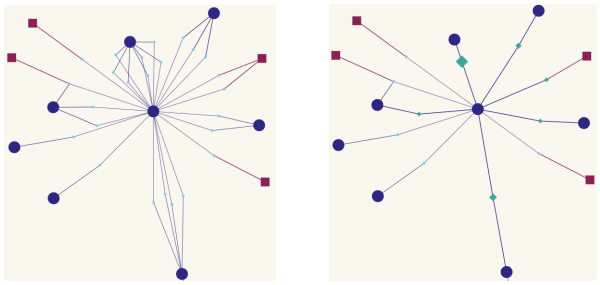
Fig. 9: Calculation Families that contain underutilized constraints viewed in a table. Clicking on the Component ID in a row redirects the user to Node-link view of that Calculation Family.

4.1.2 Node-Link Diagram

In this view, point marks signify part and constraint nodes and line marks signify edges between parts and constraints. We use hue and size to distinguish between parts and different types of constraints (See Section 3.3). Different shapes (circle, diamond, square) are also used to encode the different constraint types so that they could be more easily identified by the user. We encode the candidate constraint for removal using a designated hue so that it stands out in the graph, and use dashed edges to represent its outgoing links. We include a collapsible legend in this view to aid first-time users (See Fig. 7). Hovering over a node or an edge displays the label of the node or edge. The user can zoom-in to specific areas of the graph by scrolling the mouse wheel.

4.1.3 Part Contraction

Some parts are redundant and could be contracted to reduce family size and occlusion. There may exist parts that are similar in the sense that they are connected to the same set of constraints. These "identical" part nodes can be contracted to form a single meta-node (See Fig. 10). The weight of the edge between the new meta-node and the constraint c_i remains u_{c_i} (since it was equal across all identical parts). The user can contract all part nodes in the graph by clicking a button. Instead of precomputing this contraction and automatically displaying the contracted representation to the user, we decided to have the user manually perform this operation. This was done to ensure that the user is aware of the total number of parts that exist in the graph, and how many parts are contracted to produce each-meta node. In addition, we encode the number of parts that were contracted to produce a meta-node using the size of the meta-node.



(a) A calculation family (b) A contracted calculation family

Fig. 10: Part Contraction.

4.1.4 Force-Directed Placement

When clicking on a Calculation Family in the Family Selector (Section 4.1.1), the user is redirected to a node-link view of that family. The layout for this view is precomputed for each family using the Scalable Force Directed Placement (SFDP) algorithm [9]. However, once the user contracts all parts in the graph, the resultant graph may still contain some occluded nodes and edges. So, we've implemented dynamic layout options. These options allow the user to dynamically lay out the contracted graph using the SFDP or the Reingold-Tilford (RT) tree layout [21] algorithms. In addition, the user can choose to use the RT tree layout with a polar coordinate post-transformation, to produce a circular tree layout. Finally, we allow the user to manually rotate the NL view. See Figures 11 - 13 for an illustration.

4.1.5 Adjacency Matrix

In certain cases, it is not evident whether the removal of the candidate constraint will produce two subgraphs of approximately equal size. To help the user quantify the relative sizes of the induced subgraphs, we allow the user to toggle a juxtaposed AM view of the contracted graph. The cells of the AM represent the outgoing edges from all constraints in the graph. We maintain a consistent colour encoding across the NL and AM views by colouring the matrix cells using the same hues of the constraints they correspond to in the graph. The ordering of the columns and rows of the matrix correspond to an ordering imposed by the y-coordinates of the constraint nodes in the Reingold-Tilford Tree layout of the graph (See Fig. 14). The AM container can be dynamically resized so that graphs with 100s of nodes can be discernable in the user's browser. However, since we do not perform any node aggregation, the AM view becomes illegible with graphs with more than 1 000 nodes. A dashed line is drawn on the Adjacency Matrix's anti-diagonal. This line aims to help the user gauge the relative sizes of the subgraphs induced by the removal of the candidate constraint. If the candidate constraint is close to or lies on the anti-diagonal, then the removal of this constraint will produce subgraphs of roughly equal size.

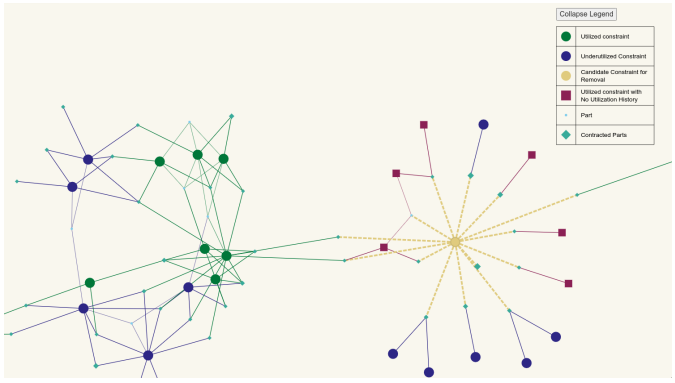


Fig. 7: A node-link view of a calculation family with a visible legend.

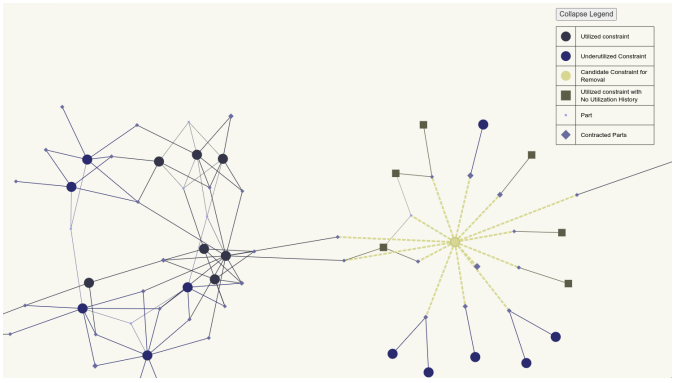


Fig. 8: The same calculation family from Fig. 7 visualized through a filter that simulates Protanopia colour blindness. The colour palette that was used to encode the types of nodes in the graph maintains a visible difference between the different types of constraints.

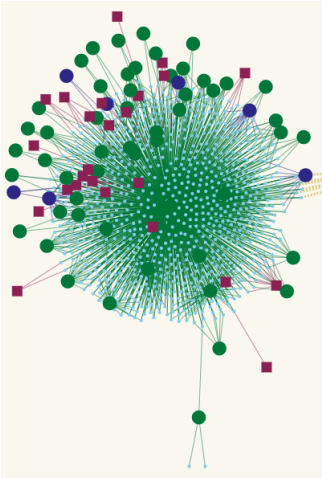


Fig. 11: An uncontracted subgraph in UCoD.

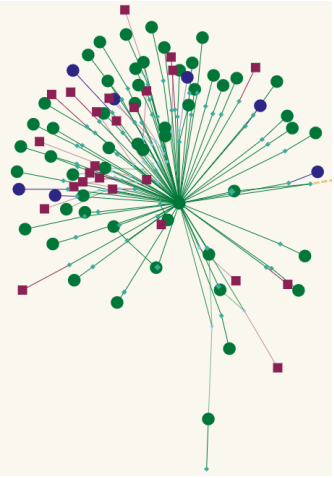


Fig. 12: A contracted subgraph in UCoD. Note that many constraints are occluded.

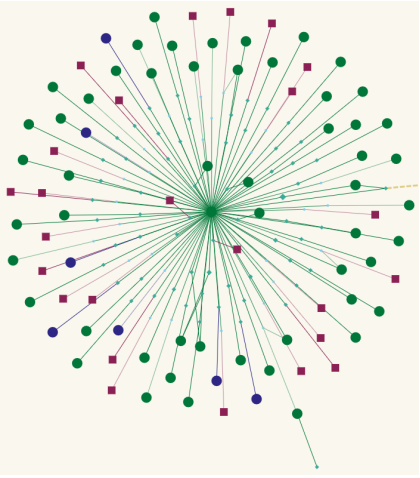


Fig. 13: The subgraph in UCoD following the application of the SFDP algorithm to the subgraph in Fig. 12. Node occlusion and edge crossings are minimized.

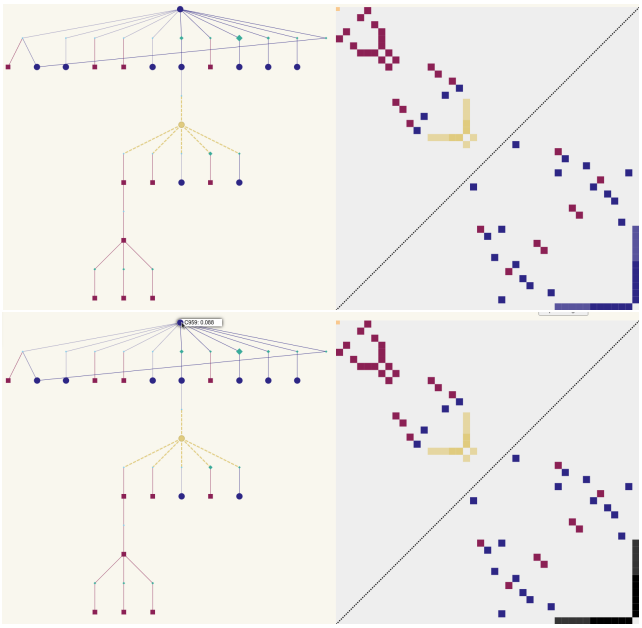


Fig. 14: A node-link view juxtaposed with the adjacency matrix view of a contracted calculation family. One-directional linked highlighting allows the user to map the nodes in the NL view to the corresponding nodes in the AM. When a node is hovered over in the NL diagram, the same node is highlighted in black in the AM.

4.1.6 Superimposed line chart

To identify constraints that are no longer being utilized, or compare utilization histories across multiple constraints, the user must be able to view the utilization time-series associated with the constraints in the NL view. To do this, the user can click on a constraint node in the graph to toggle a superimposed line chart view. By clicking on a constraint, that constraint becomes active. This is denoted by a unique shape encoding of a cross with a black node border. By holding the spacebar and clicking on additional constraints, the active constraints are superimposed in the line chart view. Each line chart that is displayed is encoded using a unique shape (either a circle, cross, rhombus, square, pentagram, triangle, or a Y shape) and colour (using the Tableau 10 colour palette [24]). Each time-series observation is encoded using the shape and hue for that line, and the observations are connected using a line mark with the same hue. The values for the shared vertical axis span the range $[0, 1]$, since all time-series represent the constraint utilization ratio. The horizontal axis is constructed using the union of all the dates in the active constraints' time-series.

By hovering over a specific constraint, the opacity for all other lines in the chart is reduced, and the time-series that corresponds to that constraint is highlighted. This allows the user to focus on the utilization history of a constraint of interest, while still being able to compare its trends of utilization to the other selected constraints. See Figures 15 - 16 for an illustration.

4.2 Implementation

1. **Data ingestion and preprocessing**: The preprocessing pipeline was implemented using the `pandas python` module [18] and was reused from our previous work. This included removing duplicate constraint utilization observations, and the detection of underutilized constraints using the recursive minimum cut algorithm.

2. **Graph construction and layout**: Using the tables described in Section 3.1, we used the `igraph python` [5] module to construct the node-link representation of all calculation families. To precompute the SFDP layout of calculation families, we used the command-line interface for `graphviz` [6]. To allow the user to dynamically lay out the node-link view, we implemented a `flask` [22] server that could ingest the coordinates of the nodes in the NL view, calculate the coordinates in the chosen layout, and output the new coordinates to the frontend in the Javascript Object Notation (JSON) format.

3. **NL View**: we bootstrapped our implementation for the NL view using `sigmajs` [11]. We also adapted some of the node selection functionality from an active fork of the `sigmajs` repository [8].

4. **AM, Superimposed Line Chart Views**: The interactive AM and line chart views were implemented using `D3` [4]. We adapted `D3` blocks that produced an adjacency matrix [25] and multiline chart [13] for use in UCoD.

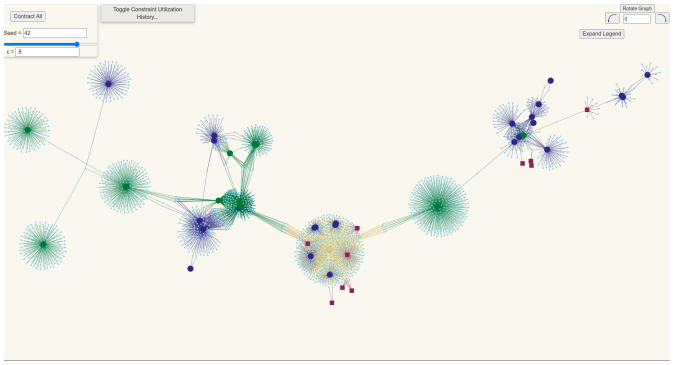
5 Scenario of Use

Andrea is a Supply Chain Planner at Kinaxis. She has been tasked with the integration of a new client's supply chain into Kinaxis' concurrent planning platform. Given the constraint utilization history and projections in the customer's product structure, she constructs a dataset of Calculation Families. However, on the first few iterations of the scheduling calculations, she notes that the calculations take too long. Andrea suspects that some of the calculation families in the dataset might be too large. She wonders whether the client has erroneously included underutilized constraints in certain families, and whether removing said constraints would speed up the parallel scheduling calculations. To find out whether this is true:

1. Andrea uploads the new client's Calculation Family dataset to UCoD.

2. The recursive minimum-cut removal algorithm returns that Family f_1 contains an underutilized constraint whose removal from the graph will split up the family substantially.

3. Andrea selects this family in UCoD to visualize it.



4. Andrea is unable to view the occluded and unordered clusters of constraints and parts in the family. She decides to contract all part nodes in the graph, and apply the `sfdp` layout to the graph.

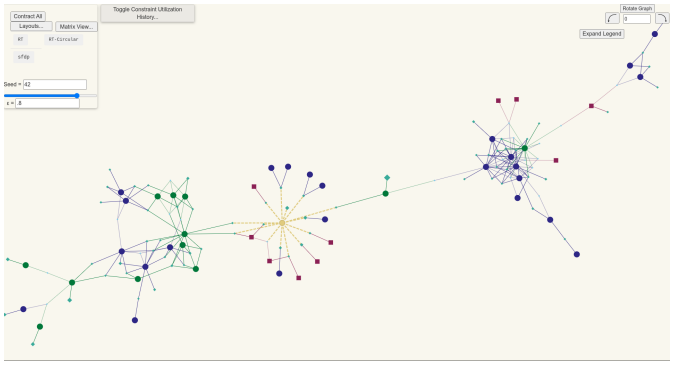


Fig. 15: A superimposed line chart that represents the utilization time-series for the active constraints in the NL view. The order of selection of active constraints may cause occlusion: the time-series for constraints that were selected more recently occlude the time-series for constraints that were selected before.

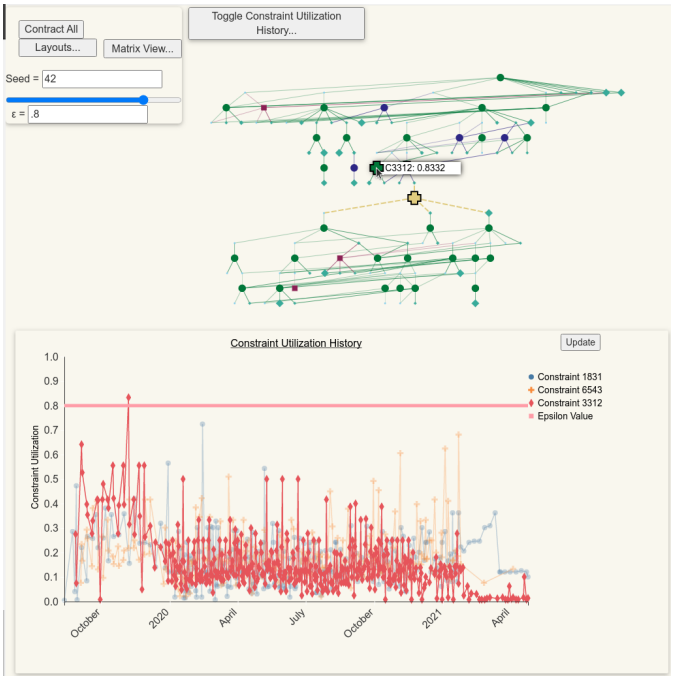
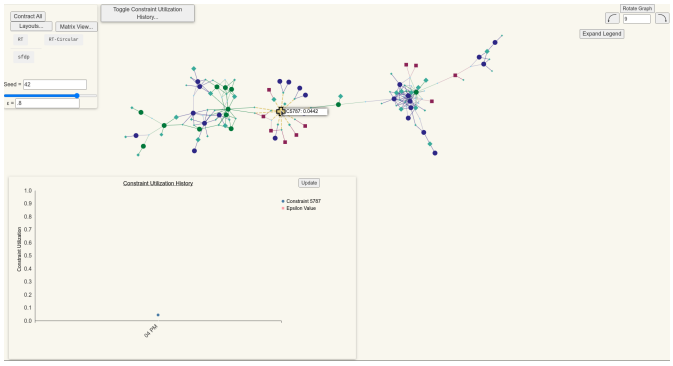
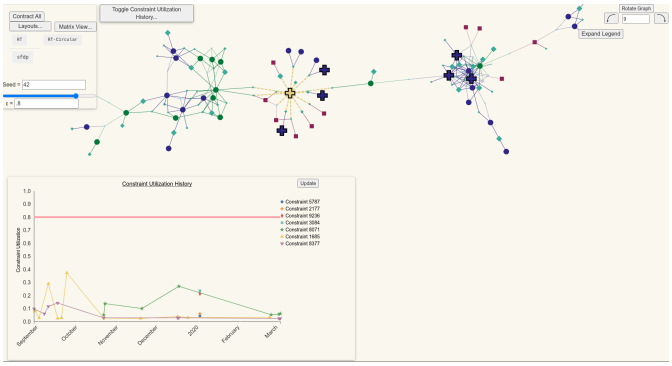


Fig. 16: Hovering over a constraint in the NL view highlights that constraint's time-series in the superimposed line chart.

5. Andrea clicks on the Candidate Constraint for Removal, and notes that its utilization history consists of a single observation. It could be that this constraint was introduced into the calculation family erroneously, or that it was only required for the scheduling of this product structure on a specific date, and was not subsequently removed from the graph.



6. To find out why the candidate constraint was introduced to the product structure, and whether it can be removed, Andrea highlights nearby constraints in the graph to view their utilization histories in the superimposed line chart. She notes that the candidate constraint's neighbouring nodes were all utilized on the same date.



7. Using this information, Andrea decides to remove the underutilized constraint from the graph. This removal partitions the graph into separate components, and so will optimize the scheduling of this product structure. However, having removed this constraint, she will continue to monitor this product structure to verify that it continues to represent a valid plan.

6 Limitations

To display the utilization time-series for a constraint, that constraint must be selected by the user. When selected, an active constraint is identified by a cross shape with a black border. A clearer way to highlight the set of active constraints would have been to set their node labels as fixed while they are selected. We were not able to implement this functionality using the `sigmajs` Application Programming Interface. Currently, hovering over nodes and edges displays their label, but once we select a constraint to display its utilization time-series, its label disappears.

Using an additional colour palette for the superimposed line chart introduces two limitations to our design. First, colour-blind individuals will have difficulty in distinguishing certain hues in the Tableau10

Colour Palette. Second, the new colour encoding for the constraints is not maintained across the line chart and NL views; selecting a constraint to display in the line chart does not change the constraint's node colour in the node-link view. This may result in an unclear mapping between active nodes and their corresponding line charts. We attempted to address this ambiguity by including a legend in the line chart, and implementing one-directional linked highlighting between the NL diagram and the line chart.

The AM view and it's anti-diagonal are shown to the user to compare the sizes of the two subgraphs that are induced by the removal of the candidate constraint. It would have been beneficial to explicitly include the sizes of the product structure and the two subgraphs in the visualization, to highlight the relative sizes of the subgraphs.

Finally, UCoD currently does not *fully* support linked highlighting across all its views. Hovering over a constraint node in the NL view highlights that node in AM view and the constraint utilization time-series in the superimposed line chart. However, hovering over a node in AM view does not highlight that node in the NL view, and hovering over a line in the line chart does not highlight its corresponding constraint. Bidirectional linked highlighting would be especially valuable in the context of the AM. Since the rows and columns of the AM are unlabeled, the user is unable to map nodes in the AM to their correspondents in the NL view. To identify a node in the AM, the user needs to selectively hover over nodes in the NL view, until the node to be identified is highlighted in the AM. When dealing with product structures with more than 100 nodes, this is tedious and possibly infeasible.

7 Future Work

First, we've established that the current implementation of the AM view in UCoD is not suited to display product structures with more than 1000 nodes. However, in certain cases, product structures can contain clusters of tightly connected parts and constraints. These clusters can be detected using community detection algorithms [3, 26] that produce hierarchical community separations given an input graph, and assign each node in the graph to a community. This hierarchy can be used to simplify the AM view. After assigning each node to its detected community, we could contract all nodes in the same community to produce meta-nodes, and visualize the aggregated, smaller graph using the adjacency matrix. To "zoom-in" on the communities denoted by the meta-nodes, users could click on their corresponding entries in their adjacency matrix. Previous work has produced interactive visualizations of the hierarchy trees of graphs [1], so we would be interested in exploring the benefits of computing a hierarchical clustering that could simplify the representation of product structures.

Second, due to the time constraints imposed by the course, the use of the superimposed line chart was not justified with a user study. In certain cases, highlighting multiple constraints with overlapping time-series can result in illegible line charts (See Fig. 15). Future work should empirically compare this design choice with juxtaposed filled area charts to assess the effectiveness of each idiom in the context of the tasks of outlier detection, trend identification, and constraint utilization time-series comparison.

8 Lessons Learned

We attempted to use the concepts and idioms we learned in class and the textbook to develop a reasonable solution to a domain-specific problem. One concept that required careful iteration was the choice of an unambiguous visual encoding. Specifically, we experimented with different colour palettes for the nodes in the graph while attempting to strike a balance between a high level of contrast between categories of constraints, and using a colourblind accessible palette. This consideration was further complicated when we introduced the superimposed line chart view, which required the addition of a new colour palette to distinguish between different time-series.

Another important lesson we'll take away from this work is the critical importance of formulating a concrete and well-researched implementational plan in the first phases of a design study. At the beginning of the project, we had a *rough* outline of the frameworks we will use to implement the features we thought would be useful to provide in UCoD. However, we adapted our visual encoding in the process and replaced certain features we thought would be useful for users with others. This forced us to explore the landscape of tools, packages, and languages that we could use for our purposes near the end of the course, which was stressful and time-consuming. Still, this exploration exposed us to the extensibility and ubiquity of `D3`, which we are sure we will continue to use for information visualization in our future computer science research.

9 Conclusion

In UCoD, a supply chain calculation family is represented using a juxtaposed node-link plot and an adjacency matrix view. Our tool enables supply chain planners to interactively explore utilization trends and anomalies of the constraints in the graph. Each product structure visualized in UCoD is known to contain a candidate constraint for removal: a node that is consistently underutilized, and whose removal from the graph will disconnect the graph into roughly equal sizes. UCoD highlights candidate constraints using dashed edges with a distinct hue for ease of identification by the user. Selecting constraints in UCoD toggles the utilization time-series for the selected constraints. Together, these capabilities allow supply chain planners to visualize the topologies of their product structure and analyze underutilization in the constraints of their product structure. Finally, by removing underutilized constraints from their calculation families, planners can optimize the parallel scheduling of their supply chains.

10 MILESTONES

10.1 Completed Work by Alex Trostanovsky

Name	Description	Owner	Estimate	Actual	Completed On
Project Pitch	1. Prepare Presentation slides.	Alex	6 hours	6 hours	Oct. 1
Proposal write up	1. Brainstorm different sections, and deliver the document.	Alex	4 hours	2 hours	Oct. 23
Interactive Constraint Contraction	1. The user can contract all underutilized constraints by clicking `Contract All`	Alex	16 hours	16 hours	Nov. 15
Adjacency Matrix View	1. The node-link view of a calculation family is accompanied by the adjacency matrix view.	Alex	16 hours	12 hours	Nov. 22
Data/Graph Input Handler	1. The families containing the candidate constraints for removal are displayed to the user.	Alex	10 hours	8 hours	Nov. 15
Flask backend	1. Create a server that will calculate graph layout using `igraph` and `graph-tool`	Alex	10 hours	8 hours	Nov. 15
Superimposed line Chart	1. The user is able to highlight active nodes and view their constraint utilization history in a collapsible view	Alex	10 hours	16 hours	Dec. 9
Proposal Update	1. Revise project report based on notes	Alex	4 hours	8 hours	Nov. 18
Presentation Preparation	1. Prepare Presentation slides 2. Record and edit final video	Alex	8 hours	16 hours	Dec. 10
Final Report	1. Complete the final report	Alex	8 hours	12 hours	Dec. 14

Table 1: Alex's Project Milestones

10.2 Completed Work by Nikola Cucuk

Name	Description	Owner	Estimate	Actual	Completed On
Project Pitch	1. Airbnb booking visualization, prepare Presentation slides. 2. Plot the free GIS data-set for Vancouver.	Nikola	6 hours	10 hours	Oct. 1
Proposal write up	1. Related work and Milestones section	Nikola	5 hours	8 hours	Oct. 23
Environment setup	1. Setup the environment described in Section 4.2: `Python`, `JavaScript`, `GraphViz`	Nikola	2 hours	7 hours	Oct. 25-Nov.31
Adjacency Matrix View	1. Plot the DOT language file format as an Adjacency Matrix	Nikola	8 hours	18 hours	Nov. 22
MinCut Node/edges viz enhancement	1. Sweep for the Candidate Constraint in each family, extract nodes and edges 2. Change the nodes' and edges' appearance.	Nikola	6 hours	18 hours	Nov. 26
Proposal Update	1. Revise project report based on notes	Nikola	4 hours	8 hours	Nov. 18
Presentation Preparation	1. Prepare Presentation slides 2. Record video sections	Nikola	8 hours	13 hours	Dec. 10
Final Report	1. Complete the final report	Nikola	7 hours	12 hours	Dec. 14

Table 2: Nikola's Project Milestones

REFERENCES

[1] Daniel Archambault, Tamara Munzner, and David Auber. Grouseflocks: Steerable exploration of graph hierarchy space. *IEEE Trans. Visualization & Comp. graphics*, 14(4):900–913, 2008.

[2] William Z Bernstein, Devarajan Ramanujan, Niklas Elmqvist, Fu Zhao, and Karthik Ramani. Viser: Visualizing supply chains for eco-conscious redesign. In *Intl. Design Eng. Technical Conf. and Computers and Information in Eng. Conf.*, volume 46353, page V004T06A049. American Society of Mechanical Engineers, American Society of Mechanical Engineers (ASME), 2014.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech.: Theory and Experiments*, 2008(10): P10008, 2008.

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics*, 17(12):2301–2309, 2011.

[5] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems: 1695, 2006. URL `https://igraph.org`.

[6] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 30(11):1203–1233, 2000.

[7] Tera Marie Green, William Ribarsky, and Brian Fisher. Visual analytics for complex concepts using a human cognition model. In *2008 IEEE Symp. Visual Analytics Science & Technology*, pages 91–98. IEEE, 2008.

[8] Sébastien Heymann. *linkurious*, 2020 (accessed November 1, 2020). URL `https://github.com/Linkurious/linkurious.js/tree/develop`.

[9] Yifan Hu. Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[10] Zhi-Hua Hu, Bin Yang, You-Fang Huang, and Yan-Ping Meng. Visualization framework for container supply chain by information acquisition and presentation technologies. *J. Software*, 5(11): 1236–1242, 2010.

[11] Alex Jacomy. *sigmajs*, 2020 (accessed November 1, 2020). URL `http://sigmajs.org`.

[12] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, 9(6):e98679, 2014.

[13] Ziggy Johnson. *d3.legend example*, 2020 (accessed November 15, 2020). URL `http://bl.ocks.org/ZJONSSON/3918369`.

[14] Ying-Jer Kao, Yun-Da Hsieh, and Pochung Chen. Uni10: An open-source library for tensor network algorithms. In *J. Physics: Conf. Series*, volume 640, page 012040, 2015.

[15] Mohamad Kassem, Nashwan N Dawood, Claudio Benghi, Mohsin Siddiqui, and Donald Mitchell. Coordinaton and visualization of distributed schedules in the construction supply chain: A potential solution. In *10th Intl. Conf. Construction Applications of Virtual Reality*, pages 77–86. CONVR2010 Organizing Committee, 2010.

[16] Sergio Lazzarini, Fabio Chaddad, and Michael Cook. Integrating supply chain and network analyses: the study of netchains. *J. Chain & Network Science*, 1(1):7–22, 2001.

[17] Michael J McGuffin. Simple algorithms for network visualization: A tutorial. *Tsinghua Science and Technology*, 17(4):383–398, 2012.

[18] Wes McKinney. Data structures for statistical computing in python. In *Proc. 9th Python in Science Conf.*, pages 51–56, 2010.

[19] Shotaro Minegishi and Daniel Thiel. System dynamics modeling and simulation of a particular food supply chain. *Simulation practice and theory*, 8(5):321–339, 2000.

[20] Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.

[21] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Trans. Software Engineering*, (2):223–228, 1981.

[22] Armin Ronacher. *Flask Documentation (1.1.x)*, 2020 (accessed November 1, 2020). URL `https://flask.palletsprojects.com/en/1.1.x/`.

[23] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *JACM*, 44(4):585–591, 1997.

[24] Maureen Stone. *How we designed the new color palettes in Tableau 10*, 2016 (accessed December 1, 2020). URL `https://www.tableau.com/about/blog/2016/7/colors-upgrade-tableau-10-56782`.

[25] Micah Stubbs. *adjacency matrix layout*, 2020 (accessed November 15, 2020). URL `https://bl.ocks.org/micahstubbs/7f360cc66abfa28b400b96bc75b8984e`.

[26] Vincent A Traag, Ludo Waltman, and Nees Jan van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 9(1):1–12, 2019.

[27] Frank Van Ham, Hans-Jörg Schulz, and Joan M Dimicco. Honeycomb: Visual analysis of large scale social networks. In *IFIP Conf. HCI*, pages 429–442. Springer, 2009.

[28] Martin Wattenberg. Visual exploration of multivariate graphs. In *Proc. SIGCHI Conf. Human Factors Computing Systems*, pages 811–819. Morgan Kaufmann Publishers Inc, 2006.