Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Forschungsarbeit Informatik MSc.

On linear layouts of graphs with SAT

Mirco Haug

Jul 11, 2019

Reviewers

Michael A. Bekos
(Informatik)
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Michael Kaufmann
(Informatik)
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Mirco Haug:
On linear layouts of graphs with SAT
Forschungsarbeit Informatik MSc.
Eberhard Karls Universität Tübingen
Thesis period: April 2019 - July 2019

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Arbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift

Abstract

During the course of this research project an application to compute linear layouts of graphs was created. This application provides a wide variety of options and constraints to force the resulting layout in the desired shape. The application is based on python and flask. The actual computing is done by creating SAT instances from the given problem and pass those instances to the lingeling SAT solver. The application provides a extensively documented REST API to create and read problem instances. This API is currently only used by the front end located at http://algo.inf.uni-tuebingen.de/linearlayouts/ . But in principle everybody is able to implement their own front end to accommodate their needs.

Zusammenfassung

Im Verlauf dieses Forschungsprojekts wurde eine Anwendung entwickelt welche lineare Layouts von Graphen berechnen kann. Die Anwendung stellt eine große Bandbreite an Optionen und Bedingungen zur Verfügung, mithilfe derer es möglich ist das errechnete lineare Layout in die gewünschte Form zu zwingen. Die Anwendung basiert auf Python und Flask. Die Errechnung des linearen Layouts basiert darauf, dass die gegebene Probleminstanz zuerst in ein SAT Problem umgewandelt wird. Danach wird die SAT Instanz an den SAT Solver "lingeling" übergeben. Das Ergebnis wird zurück umgewandelt und dem Benutzer zur Verfügung gestellt. Die Anwendung stellt eine vollständig dokumentierte REST API zur Verfügung um Probleminstanzen anzunehmen. Diese Schnittstelle wird zur Zeit nur von der front end http://algo.inf.uni-tuebingen.de/linearlayouts/ verwendet. Prinizipell können jedoch beliebige andere andere Frontends mit eigenen Schwerpunkten für diese Schnittstelle erstellt werden.

Contents

List of Figures

Chapter 1

Motivation

The goal of this application is to find linear layouts from given graphs under the restriction of various constraints. There is an already existing tool implemented by [Pup] . But this tool is limited to finding general linear layouts, with no constraints regarding individual pages or even individual graph elements. The problem is that many proofs and scientific projects about linear layouts require a specific structure of the resulting linear layout. If such a proof has to be investigated with the previous mentioned tool, it would take a lot of time to create the graph which produces a linear layout as required. This problem is intensified by the fact that for most graphs more than on linear layout exists of which possibly only one is interesting.

This application will help to omit this graph finding phase and directly restrict an arbitrary linear layout to the desired properties if they exist. For this reason the a dedicated front end will be developed which allows graph manipulation and attaching constraints to all graph elements. This application will then accept a problem definition consisting of the graph, the arguments for the linear layout, like pagenumber and page type, and a list of additional constraints.

To make this application even more useful there will be no dependencies on the used front end. Meaning anybody can use the server from whichever front end suits their need as long as they honor the API contract.

Chapter 2

Technology basics

This chapter describes the technology and the frameworks used in the project.

2.1 REST APIs

REST Interfaces or APIs are meant to provide an interface to other programs. Instead of the HTML[1] files normal web services deliver, a web service providing a REST interface delivers files in the JSON[2] format. Like a normal web server is providing different sites at different sub URLs, see the footnotes, a REST service can also provide different data at different sub URLs.

In addition to different URLs a REST API can also use the different HTTP verbs[3] to do different things. Per convention a request with the verb GET does read some data from the server, whereas a request with the verb POST does write data to the server. One combination of sub URL and HTTP verbs does identify a so called endpoint.

Each endpoint in a REST service has a specified JSON format in which it accepts data and a specified format in which it returns data.

The external Interface this application provides is a REST interface. For an detailed information which endpoints are available and which format they use see the root page of the server.

2.2 Python

This application is implemented in python.

“Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python Web site, https://www.python.org/, and may be freely

[1] https://en.wikipedia.org/wiki/HTML
[2] https://en.wikipedia.org/wiki/JSON
[3] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation.

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications." [Fou19]

2.3 Flask and Flask-restplus

Flask is a python framework which enables a Developer to write REST services with python.

Flask-restplus is another framework which enables the Developer to easily define the JSON format of each created REST endpoint.

With these two Frameworks the external interface of the service is implemented. The implementation of the interface is mainly done in the class `App`

2.4 SQlite

SQLite is a database engine which does not need a server. Instead all the work a database server normally does is included in the SQLite client. SQLite does provide a fully featured SQL[4] interface. The data is stored in a single file in the SQLite file format. Typically those files have the extension *.db*. Python already contains such a SQLite client.

The application stores the computed results in such a SQLite database.[5] The class encapsulating the database access from the rest of the application is `DataStore`.

2.5 The boolean satisfiability problem

The boolean satisfiability problem is the problem of finding a interpretation which satisfies a given boolean formula. For simple formulas such as the following, this is rather trivial.

$$A \land B \land C$$

But as the formula grows and introduces more variables and clauses the possible solution space grows strongly and therefor the time to find a satisfying interpretation of the formula grows. The "boolean satisfiability problem" or short "SAT Problem" is NP-complete.

SAT Solvers try to find an interpretation of a given formula by using computer science and optimized algorithms. There are also competitions for the best SAT Solver. In order to make things more easy the formula for a solver hast to be in the conjunctive normal form (CNF)[6].

> Small problem instances handled by this application contain roughly 3000 variables and 80000 CNF clauses and are solved within 0.1 seconds.

In order to achieve the Goals mentioned in *Motivation*, the application does formulate the problem as a boolean formula and passes this formula to the SAT Solver.

[4] https://en.wikipedia.org/wiki/SQL
[5] https://www.sqlite.org/index.html
[6] https://en.wikipedia.org/wiki/Conjunctive_normal_form

The translation of the problem in a boolean formula and back is task of the class `SatModel`. The actual solving of this formula is passed to the SAT Solver lingeling.

2.6 Setup project

This project requires the *lingeling* binary present on the system. Currently lingeling does only support UNIX operating systems. So the application only runs in UNIX environments.

Fortunately also unix like environments like cygwin[7] or Windows Subsystem for Linux (WSL)[8] are able to run lingeling. The application itself does not depend on UNIX but only on python which is (almost) platform independent.

For detailed information on how to build and run the project see the README.md file.

[7] https://www.cygwin.com/

[8] https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux

Chapter 3

Theoretical baseline

This chapter first describes linear layouts and the different types of linear layouts. Afterwards it provides insight in how the constraints from the linear layout itself and the additional constraint are encoded with SAT.

3.1 Linear layout

A linear layout of a graph simply states that all nodes/vertices are on one line, sometimes called spine. Each edge can be assigned to a page. The additional constraints are that ,depending on the type of the linear layout, there have to be no crossings of edges which are on the same page. The combination of node order and edge assignment causes a big problem space for this problem.

The two types of linear layouts used in the application are book embeddings and queues. The sample graph to demonstrate this two types will be the Goldner–Harary graph shown in Fig. 3.1.

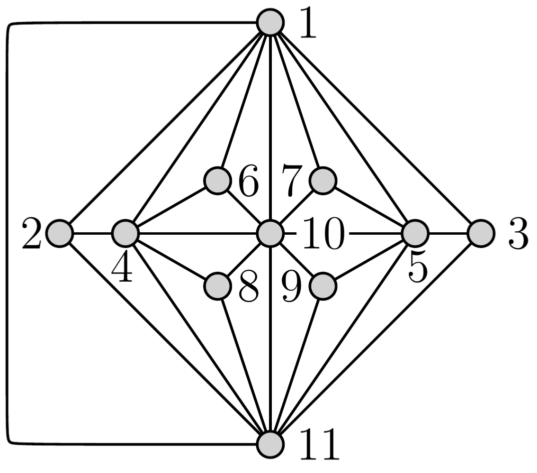

Fig. 3.1: Goldner–Harary graph

3.1.1 Book embedding

A book embedding layout is quantified by the number of pages an graph needs for its book embedding. The directive of the book embedding is, that no two edges of the same color intersect. Therefor all edges represent a stack. The given graph Fig. 3.1 as book embedding can be seen in Fig. 3.2.

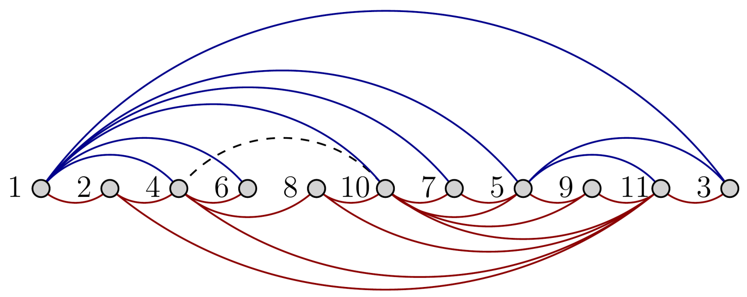

Fig. 3.2: Goldner–Harary graph as book embedding / stack

3.1.2 Queue embedding

A queue embedding layout of the graph shown in Fig. 3.1 is show in Fig. 3.3. A Queue embedding is subject to the constraint that not two edges of one color do completely enclose each other.

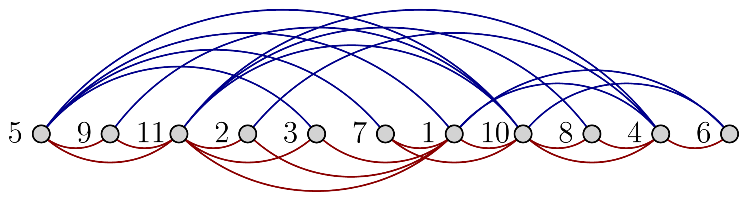

Fig. 3.3: Goldner–Harary graph as queue

Fig. 3.4 shows a summary on allowed patterns within linear layouts.

3.2 Encoding with SAT

The in *Motivation* defined problem gets hard to solve because of the big solution space. SAT Solvers are used to big problem spaces and can therefor find solutions in reasonable times. The application translates the problem of finding an linear layout into a SAT problem and lets it be solved by specialized SAT solvers. This chapter describes how the problem is encoded with SAT.

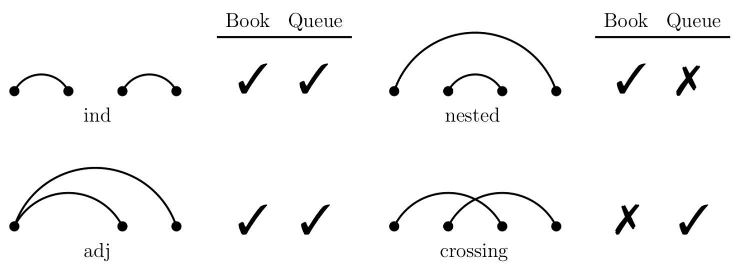

Fig. 3.4: Summary on linear layouts taken from [Wol18]

3.2.1 Linear layout

The basic parameters of a layout like node order or edge assignment are encoded according to chapter two of this paper [BKZ15] .

Let $G = (E, V)$ with $V = \{v_1, v_2, \cdots, v_n\}$ and $E = \{e_1, e_2, \cdots, e_m\}$. p denotes the index of the page and P denotes all pages.

The node order is then defined as $\sigma(v_i, v_j) \quad \forall v_i, v_j \in V$ with pairwise distinct i,j. For σ asymmetry and transitivity have to hold.

The edge to page assignment of edge i to page p is denoted by $\phi_p(e_i)$. Clearly has $\phi_1(e_i) \vee \cdots \vee \phi_p(e_i) \forall e_i \in E$ to hold to assign each edge to at least one page.

For each STACK page p the following clauses are added to forbid alternating patterns of vertexes of e1 and e2 if all vertexes are distinct.

$$\begin{aligned}
&\phi_p(e_1) \wedge \phi_p(e_2) \implies \neg(\sigma(e_1n_i, e_2n_k) \wedge \sigma(e_2n_k, e_1n_j) \wedge \sigma(e_1n_j, e_2n_l)) \\
&\qquad \text{w.r.t:} \\
&\qquad\qquad \{e_1n_i, e_1n_j\} \in V(e_1), \{e_2n_k, e_2n_l\} \in V(e_2) \\
&\qquad\qquad e_1n_i \neq e_1n_j \neq e_2n_k \neq e_2n_l \\
&\qquad\qquad \forall e_1 \neq e_2 \in E
\end{aligned}$$

For QUEUE pages q there are clauses added which forbid enclosing patterns.

$$\begin{aligned}
&\phi_q(e_1) \wedge \phi_q(e_2) \implies \neg(\sigma(e_1n_i, e_2n_k) \wedge \sigma(e_2n_k, e_2n_l) \wedge \sigma(e_2n_l, e_1n_j)) \\
&\qquad \text{w.r.t:} \\
&\qquad\qquad \{e_1n_i, e_1n_j\} \in V(e_1), \{e_2n_k, e_2n_l\} \in V(e_2) \\
&\qquad\qquad e_1n_i \neq e_1n_j \neq e_2n_k \neq e_2n_l \\
&\qquad\qquad \forall e_1 \neq e_2 \in E
\end{aligned}$$

3.2.2 Page constraints

In addition to the type, each page can have an additional constraint.

The first of such constraints is the DISPENSABLE constraint which does restrict the order of each vertex to at most one. The corresponding clauses are:

$$\begin{aligned}
&\neg\phi_p(e_1) \vee \neg\phi_p(e_2) \\
&\qquad \text{w.r.t:} \\
&\qquad\qquad V(e_1) \cap V(e_2) \neq \{\} \quad \forall e_1, e_2 \in E
\end{aligned}$$

The second of such constraints is FOREST. Which enforces, that the graph on page p is acyclic. To encode this new variables have to be introduced. One type for the parent relationship and one for the ancestor relationship. The ancestor relationship is used to forbid cyclic graphs. The detailed formulation is described in chapter 2.1 of [BKZ15].

The TREE constraint is also described there. It basically adds the a additional variable to indicate if one node is the root of the tree. Following this at most one root is allowed.

The implementation of this constraints is located in `add_page_constraints()`

3.2.3 Additional constraints

The application supports a wide variety of additional constraints the user can impose on the given problem instance. In the following, the there will be a short description of the constraint in text form followed by the logical projection. The title will represent the constraint type accepted by the API. The implementation to this constraints is found in `add_additional_constraints()`.

EDGES_ON_PAGES

This constraint forces the given edges E^* to a given pages P^*. The implementation is simply:

$$\bigwedge_{p \in P^*} \phi_p(e) \quad \forall e \in E^*$$

EDGES_SAME_PAGES

This constraint will force the given edges $e_1, \cdots, e_n$ to the same page. Because there is no general way to formulate this constraint in CNF for an arbitrary number of pages it is only implemented up to four pages. The general formulation can be seen bellow.

$$\bigvee_{p \in P} (\phi_p(e_{i-1}) \wedge \phi_p(e_i)) \quad \forall i \in [2, n]$$

EDGES_DIFFERENT_PAGES

This constraint forces all given edges E^* on different pages. It will create unsolvable instances if more edges are given than there are pages. In this case the application will throw an error. The constraint is pairwise implemented as:

$$\bigwedge_{p \in P} (\neg\phi_p(e_i) \vee \neg\phi_p(e_j)) \quad \forall e_i \neq e_j \in E^*$$

EDGES_TO_SUB_ARC_ON_PAGES

This is rather special constraint. It is related to the proof sketched in [Yan86]. According to the description in the proof this constraint enforces the following: If an edge has one endpoint in one of two specifically designated nodes s, t and the other endpoint between them, then it is restricted to certain pages P^*. The logical formula is:

$$\bigvee_{p \in P^*} (\phi_p(e)) \quad \forall e \in E \mid V(e) \cap \{s, t\} \neq \{\} \wedge V(e) \in [s, t]$$

EDGES_FROM_NODES_ON_PAGES

The current constraint is related to the *EDGES_TO_SUB_ARC_ON_PAGES* constraint. Given this constraint all edges from the given nodes V^* have to be assigned to the given pages P^*.

$$\bigvee_{p \in P^*} (\phi_p(e)) \quad \forall e \in E \mid V(e) \cap V^* \neq \{\}$$

NODES_PREDECESSOR

This and the following constraints apply to nodes. The current constraint does require that one set of nodes V_1^* is before an other set of nodes V_2^*.

$$\sigma(n_i, n_j) \quad \forall n_i \in V_1^*; n_j \in V_2^*$$

NODES_REQUIRE_ABSOLUTE_ORDER

Whereas the previous constraint is only able to encode relative order, does this constraint encode absolute order. So no node is allowed in between the given node order $n_1, \cdots, n_j$. Also do they have to appear in exactly this order. The logical encoding uses exactly this trick:

$$\sigma(n_{i-1}, n_i) \wedge \left(\bigwedge_{n_x \in V \ \mid \ n_i \neq n_x \neq n_{i-1}} \neg \left(\sigma(n_{n-1}, n_x) \wedge \sigma(n_x, n_i)\right) \right) \quad \forall i \in [2, j]$$

NODES_REQUIRE_PARTIAL_ORDER

A series of *NODES_PREDECESSOR* constraint with one node in each set can be expressed with this constraint. It simply enforces, that the nodes appear in the order $n_1, \cdots, n_j$. It is therefor a less strict version *NODES_REQUIRE_ABSOLUTE_ORDER*. The formula is as simple as:

$$\bigwedge_{i \in [2, j]} \sigma(n_{i-1}, n_i)$$

NODES_FORBID_PARTIAL_ORDER

This encodes the opposite of *NODES_REQUIRE_PARTIAL_ORDER*. The constraint is satisfied as soon as two of the nodes switch their relative position. The formula is:

$$\neg \left(\bigwedge_{i \in [2, j]} \sigma(n_{i-1}, n_i) \right)$$

NODES_CONSECUTIVE

This constraint is similar to *NODES_REQUIRE_ABSOLUTE_ORDER* with two nodes but without regarding the particular order of the nodes. This constraint only requires that between the two given nodes n_1, n_2 there is no other node. The corresponding formula is:

$$\sigma(n_1, n_2) \Leftrightarrow \sigma(n_i, n_1) \vee \sigma(n_2, n_i) \quad \forall n_i \in V \mid n_2 \neq n_i \neq n_1$$

Chapter 4

Implementation

This chapter describes the architecture of the application and highlights implementation details. It will first describe how the different application components work together. Then the signatures and the code comments of the core classes are included and at the end of this chapter there will be some considerations regarding the performance optimization.

4.1 Architecture

The three main classes are:

- *`App`*: This class contains the interface definition to the outside world via the REST API.
- *`SatModel`*: This class contains the logic to generate the clauses for the sat solver.
- *`SolverInterface`*: This class wires the two previously mentioned classes together.

The following diagram shows how one request flows through the system. The schema flask validates against is created in *`create_app()`*.

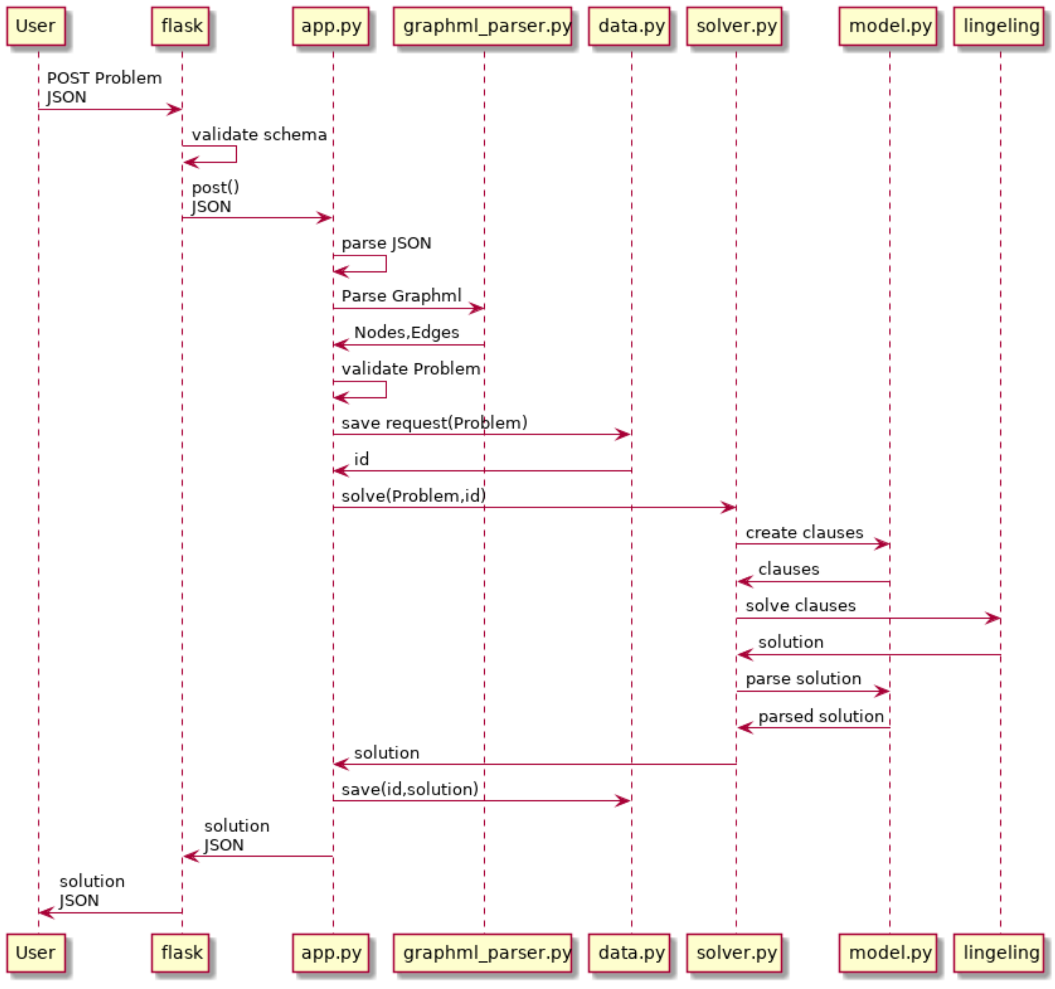

Fig. 4.1: Flow diagram of one request

4.2 Core Classes

class be.app.**App**

This class creates the entry point for the REST interface. The entry point will perform the deserialization and serialization . Also will the entry point provide some basic sanitation to the given input. Some of the checks are for example:

- Duplicated ids
- Graph size withing fairness bounds
- Structural verification

create_app() → flask.app.Flask

Initialises the the app and the api object. It adds all the provided endpoints. Also does this method define the documentation for the swagger UI and the definitions for the api object structure.

Returns the app object

`data_path = 'data.db'`
The path used to store the database file

`class` `be.model.`**`SatModel`**(*pages, edges: be.custom_types.Edge, node_ids: List[int], constraints*)
This class is responsible for generating the clauses corresponding to the given Problem instance.

`add_additional_constraints`()
Adds the clauses to encode the given additional constraints.

`add_page_assignment_clauses`()
Ensures that each edge is on at least one page.

`add_page_constraints`()
Generates the clauses to encode the page type as well as additional page constraints like DISPERSIBLE or TREE.

`add_relative_node_order_clauses`()
Ensures that asymmetry and transitivity are encoded.

`get_page_assignments_result`() → List[be.custom_types.PageAssignment]
Reads the result and translates it back to edge to page assignments.

Returns The list of page assignments

`get_vertex_order_result`() → List[str]
Reads the result and translates it back into a the computed order of vertexes.

Returns the order of the vertexes

`parse_lingeling_result`(*dimacs_string*)
Takes the result string from lingeling and parses it back into the model.

Parameters **`dimacs_string`** – the result string from lingeling

`to_dimacs_str`()
generates a string in DIMACS format encoding all the clauses. Out to conserve memory, the clauses will be deleted after the string generation.

The following module is the glue code between the *`App`*: class which handles the external interface and the *`SatModel`*: class which does the heavy lifting in creating the SAT clauses and calling the SAT solver.

`class` `be.solver.`**`SolverInterface`**
This class provides an simplified interface to the *`SatModel`*:

`classmethod solve`(*nodes*, *edges*, *pages*, *constraints*, *entity_id*) → be.custom_types.SolverResult
Initialises the class :class .*SatModel*: with the given parameters and triggers the clause generation. Afterwards the created clauses are send to the SAT solver and the result is parsed back and returned.

Parameters

- **`nodes`** – the nodes/vertexes of the problem instance
- **`edges`** – the edges of the problem instance
- **`pages`** – the pages of the problem instance

- **`constraints`** – the constraints of the problem instance
- **`entity_id`** – the id of the problem instance. This is used to wrap any exception in an `IdRelatedException` in order to pass the id to the handling method.

Returns the solved result of the problem instance

4.3 Auxiliary classes

The parsing of the graphml string happens on a low level xml basis without constructing a graph. The only validation will be to check if the nodes referenced by the edges are actually present.

The following interface is provided:

`be.graphml_parser.`**`get_nodes_and_edges_from_graph`**(*string: str) -> (typing.List[str], typing.List[be.custom_types.Edge]*)

Obtains node and enge information from a string containing a graphml definition. Ids are taken from the following hierarchy: Userdata at the xml element, id of the xml element, for edges generated from <source node>-<target node>. This hierarchy ensures that the API can use a wide variety of valid graphml as input.

Param the graphml string

Returns the lists of node ids and edges

`class` `be.data.`**`DataStore`**(*data_path*)

This class abstracts the underlying data store from the application. It provides methods create, read and update data.

`get_all`(*limit=20*, *offset=0*)

This method obtains multiple stored elements. Also provides parameters to paginate the output.

Parameters

- **`limit`** – the number of elements to return at max
- **`offset`** – the offset where to start

Returns a list of elements

`get_by_id`(*elem_id*)

Obtains an element by id.

Parameters **`elem_id`** – the element id

Returns the element or None if the id was not found.

`insert_new_element`(*element*)

Inserts a new elements into the data store and return the inserted element including the generated id.

Parameters **`element`** – the element to store

Returns the stored element

`update_entry`(*elem_id*, *element*)

Updates the entry with the given id to contain the new contents.

Parameters

- **elem_id** – the element id
- **element** – the new element content

Returns the updated element

4.4 Performance Analysis

This chapter sheds light on the particular hot spots of the application regarding technical optimization.

The Following snippet shows the time the python interpreter needed for the different methods. This was measured by the ProfilerMiddleware[9] of werkzeug[10]:

```
PATH: '/embeddings'
         2387712 function calls (2381351 primitive calls) in 4.810␣
↪seconds

   Ordered by: internal time, call count
   List reduced from 617 to 30 due to restriction <30>

   ncalls  tottime  filename:lineno(function)
     5059    1.163  {method 'poll' of 'select.poll' objects}
        1    0.868  ./SAT/server/be/model.py:72(static_to_dimacs)
        5    0.668  ./SAT/server/be/model.py:728(static_encode_page_
↪constraint_stack)
        1    0.456  ./SAT/server/be/model.py:11(static_node_order_
↪generation)
   395040    0.415  ./SAT/server/be/model.py:53(static_get_order_clauses)
   152353    0.375  {built-in method numpy.array}
    51485    0.154  ./SAT/server/be/utils.py:45(get_duplicates)
    51485    0.110  {method 'sort' of 'numpy.ndarray' objects}
   100865    0.094  {method 'tolist' of 'numpy.ndarray' objects}
   839228    0.069  {method 'append' of 'list' objects}
        1    0.068  ./SAT/server/be/solver.py:15(solve)
    51485    0.065  {method 'copy' of 'numpy.ndarray' objects}
    51485    0.046  site-packages/numpy/core/fromnumeric.py:815(sort)
   508258    0.0410 {built-in method builtins.len}
        1    0.036  {method 'translate' of 'str' objects}
        3    0.031  {method 'commit' of 'sqlite3.Connection' objects}
       12    0.030  {method 'replace' of 'str' objects}
    50097    0.018  {method 'extend' of 'list' objects}
    51485    0.013  site-packages/numpy/core/numeric.py:541(asanyarray)
     5040    0.011  {built-in method posix.write}
        1    0.003  ./SAT/server/be/solver.py:57(_call_lingeling_with_
↪string)
  3814/38    0.003  copy.py:132(deepcopy)
        1    0.003  ./SAT/server/be/model.py:359(add_page_constraints)
       21    0.002  {built-in method posix.read}
```

[9] https://werkzeug.palletsprojects.com/en/0.15.x/middleware/profiler/#module-werkzeug.middleware.profiler
[10] https://werkzeug.palletsprojects.com/en/0.15.x/

The *tottime* defines the time the interpreter ran this particular method without jumping to a sub method. The first line here *{method 'poll' of 'select.poll' objects}* is actually the waiting loop for the SAT solver to finish.

The first method which is self implemented is the call to *static_to_dimacs* which is why this method got fairly much attention in order to get optimized as much as possible.

4.4.1 DIMACS File generation

The Fig. 4.2 show the comparison of different implementations of this method. The *join* method relied heavily on the python method str.join to stick the different strings together and was the most intuitive but also the slowest. *to_str_replace* first used the build in str method of python and then replaced the various characters and strings so that the result resembles the DIMACS format. The last and best method *to_str_translate_replace* uses the same str method but then used str.translate to swap out multiple characters at once and finally str.replcae for the rest. This proved to be the best of the compared implementations.

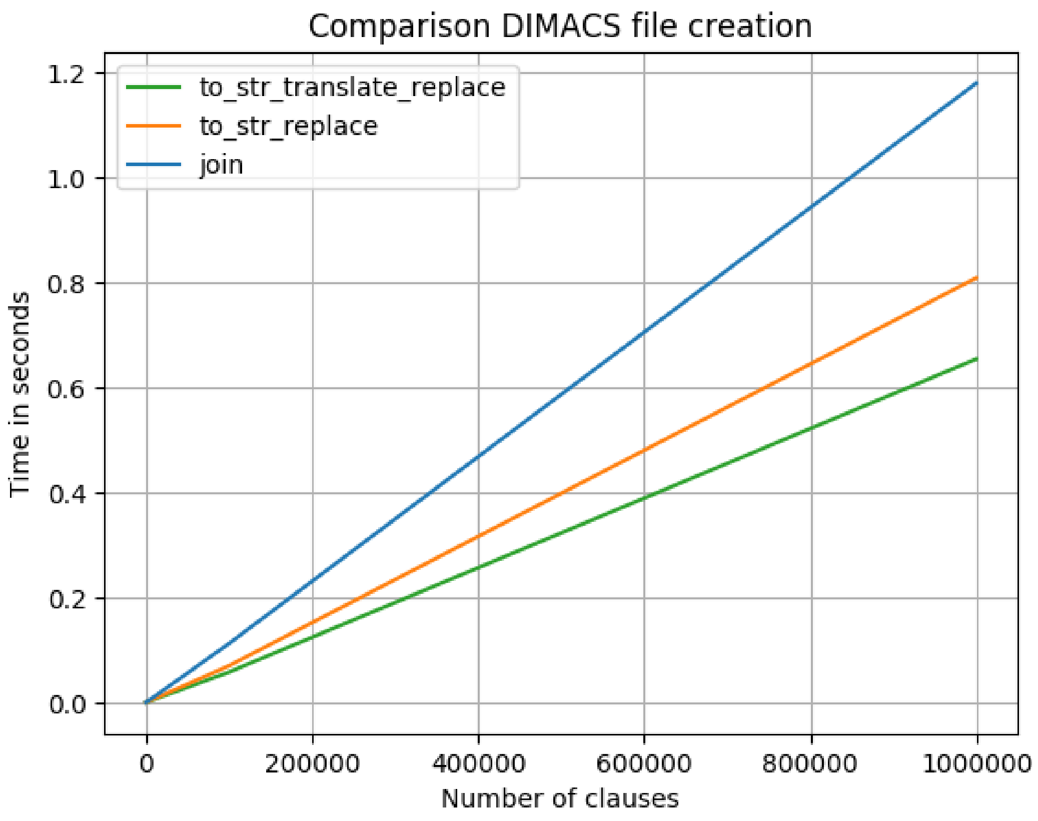

Fig. 4.2: Performance comparison of dimacs file generation

The figure shows that the time complexity of the problem is already linear and optimization can only aim to make the line less steep. This problem is ultimately not hard, but for bigger instances string sizes of several gigabytes are not unheard of and this simply takes its time.

4.4.2 Algorithms before technology

The application iterates a lot over edges or nodes. Often to create permutations of two or more edges. More often than not these permutations are not sensitive to ordering. In early iterations of this application would loop over the array as seen in the following to create permutations of 3 edges:

```
for i in a:
 for j in a:
     if i == j:
         continue
     for k in a:
         if k == i or k == j:
             continue
         # do something
         pass
```

The difference to the following more intelligent algorithm below should be obvious. Not only does the second algorithm only produce a fraction of the loops and results, but the loops which are done are all used.:

```
for i in range(len(a)):
    for j in range(i):
        for k in range(j):
            # do something
            pass
```

The difference in performance is show below.

Even by optimising the slow algorithm to run a hundred times faster it would still not beat the intelligent algorithm. Therefor the lesson is clearly optimize algorithms before technology.

4.4.3 Ahead of time

Initially the application used a logic framework like sympy to generate the CNF clauses from the definition shown in *Encoding with SAT*. This works pretty well out of the box. The problem was that the transformation to a CNF form had to be done millions of times per problem. This was realy slow. The solution was to formulate the CNF clauses in source code and use the already transformed clauses during the application. The downside of this is for example that certain constraints like *EDGES_SAME_PAGES* only work for as much pages as there are predefined clauses present.

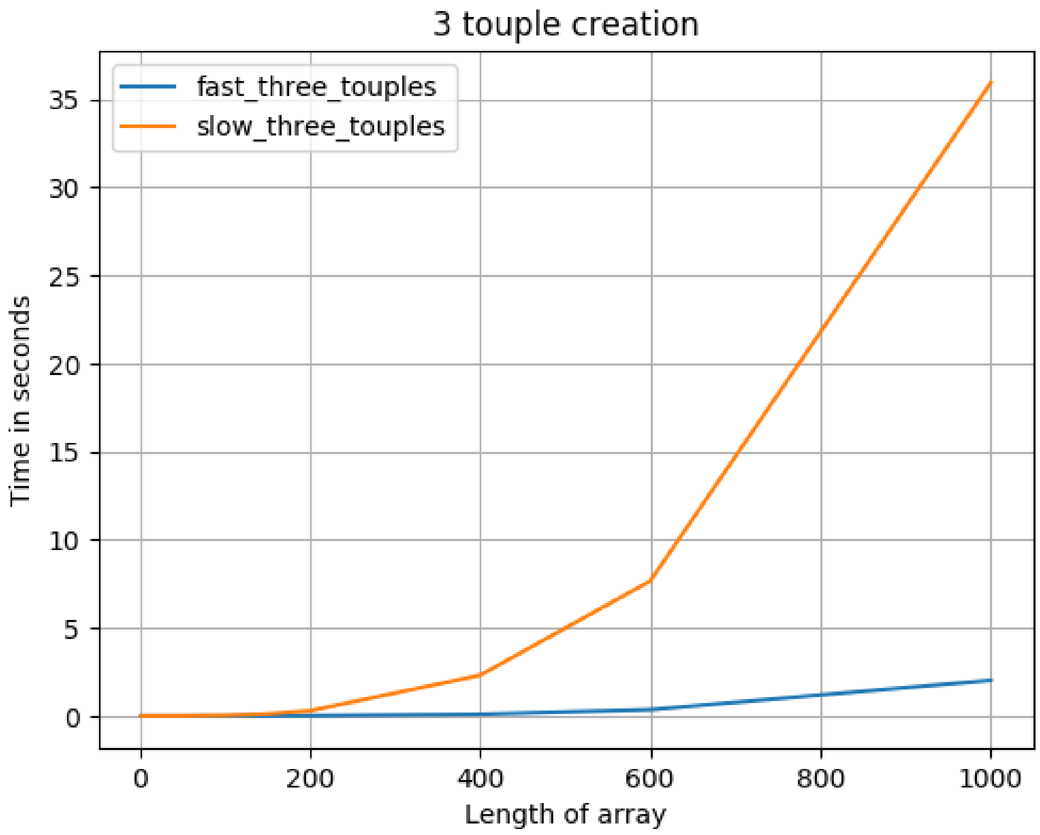

Fig. 4.3: Performance comparison of different 3-tuple generation

Chapter 5

Conclusion

The application clearly works for its intended use case. We where able to check several assumption from [Yan86] and found new insights into the problem or linear layouts. The Frontend integrates nicely with the given API and there are already several new ideas on what to check next with this application.

Bibliography

[BKZ15] Michael A. Bekos, Michael Kaufmann, and Christian Zielke. The book embedding problem from a SAT -solving perspective. In *Lecture Notes in Computer Science*, pages 125–138. Springer International Publishing, 2015. URL: https://doi.org/10.1007/978-3-319-27261-0_11, doi:10.1007/978-3-319-27261-0_11.

[Fou19] Python Software Foundation. https://docs.python.org/3.7/tutorial/index.html, 2019. [Online; accessed 10-July-2019].

[Pup] Sergey Pupyrev. Book embedding. http://be.cs.arizona.edu.

[Wol18] Jessica Wolz. Engineering linear layouts with sat. Master's thesis, Univerity of Tübingen, 2018.

[Yan86] Mihalis Yannakakis. Four pages are necessary and sufficient for planar graphs (extended abstract). In Juris Hartmanis, editor, *ACM Symposium on Theory of Computing*, 104–108. ACM , 1986. doi:10.1145/12130.12141.