Master's Thesis

Improving Usability and Robustness of Network Enrichment with KeyPathwayMiner

Author:
Martin Dissing-Hansen

Supervisors:
Dr. Jan Baumbach

Advisors:
Markus List
Nicolas Alcaraz

Syddansk Universitet
University of Southern Denmark

July 23, 2018

Legal statement

I hereby state that I wrote this thesis myself, and that I only used the sources and supporting material referenced in this thesis.

Martin Dissing-Hansen
Odense, Denmark, July 23, 2018

Acknowledgements

I would like to express the deepest appreciation to my supervisor Dr. Jan Baumbach, for inspiring me to see the world of systems biology, with the eyes of a computer scientist. I am greatly thankful for the collaboration with my advisor Markus List, for his invaluable help and inexhaustible commitment throughout the project work. I am also very thankful for the help provided by my other advisor Nicolas Alcaraz, for his help with the programming and idea development of the extensions to KPM. This project would not have reached its standard without the help of my supervisor and advisors.

I would also like to acknowledge the NanoCAN Center for their great support and the resources they made available.

A special thanks goes out to the Computational Biology research group at SDU under the lead of Jan Baumbach, with whom every week was started with smiles and support during the weekly meetings.

Getting through my thesis required more than just academic support, and I have many people to thank for listening to me, and being there for me when I needed it. Peter, Jesper, Christian, Jørgen, and especially my girlfriend Malene, who have been there for me every step of the way, helping me through this tough time.

Finally, I would like to thank my parents and my brother, for their unyielding love and support.

Abstract

Motivated by changes in the field of systems biology, a necessity to improve usability and usefulness of network enrichment analysis tools has emerged. Network enrichment analysis refers to the extraction of smaller parts of the network that are over- or under-represented for a given set of nodes. A number of network enrichment tools already exist for this purpose. One such tool is KeyPathwayMiner, which is currently available as a Cytoscape app and as a standalone version. Requirement analysis has exposed a number of shortcomings of network enrichment tools in general and KeyPathwayMiner in particular. Existing tools commonly lack robustness and validation analysis, which can be achieved through the perturbation of network graphs and validation analysis, in which the relative overlap between results and a predefined list of gold standard nodes is calculated. KeyPathwayMiner as well as comparable tools also lack compelling features for visualization of summary statistics, in which the focus is to give the user an overview of the results obtained from performing a KeyPathwayMiner run. Apart from the implementation of aforementioned features, a major goal was to improve the modularity of KeyPathwayMiner and its ability to be adapted to different user interface platforms. This was achieved by creating a single core library with which any Java user interface overlay can be used directly. Another goal was to optimally support integration with other non-Java based systems and to improve accessibility for users. This was achieved by implementing a web application interface for KeyPathwayMiner, which doubles as a web service that can be reached from all programming platforms supporting the JSON format, which was demonstrated by example of integrating KeyPathwayMiner with an R based tool for microRNA analysis. The robustness and validation analysis have proven their usefulness through tests designed to show how they can support the evaluation of the results. Finally, the web application enables users to run KeyPathwayMiner from anywhere without the need to install third party software.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In the field of systems biology, focus is shifting from single genes or proteins towards complex biological systems, which can be represented as networks that capture the topology of biological systems based on, for instance, protein-protein interactions or known gene regulations. In recent years, new high-throughput tools for data collection provided comprehensive information about interactions and components within a cell and is summarized under the collective term "OMICS"-data. Integrating biological networks with such data rich sources is a promising strategy to extract answers to concrete biological questions, and therefore there is a need for integrating multiple sets of OMICS data like gene expression, for instance, with prior knowledge captured in biological networks [17]. In this thesis, network enrichment analysis will refer to the analysis of over- or under-representation of nodes in such networks. Network analysis techniques in general are prone to be distorted by noise from different factors. One such factor could be incompleteness, where edges are biased sometimes based only on predictions without any experimental proof. Therefore, it is of the utmost importance that the results for such enrichment analysis are integrated in a wider context, partly to identify and reduce noise generated by biological or technical variations, and partly to extract an answer to a specific question or biological condition.

Several tools have been implemented to fulfill the need for such enrichment analyses. In this paper, the focus is on those implemented as Cytoscape [22] apps. For most biomedical researchers, installing and using Cytoscape can be challenging, partly because of its complex user interface, and partly due to the lack of pre-processing support, e.g. to combine expression datasets. The latter is due to the fact that pre-processing data currently needs extensive statistical knowledge, and compelling programming/scripting capabilities. The state of the art network enrichment tools are using efficient, customizable algorithms to perform analyses on biological raw data. Even though the current tools fulfill some needs of bio-medical researchers, they currently still lack features to determine the robustness of its results. The implementation of such novel robustness features would vastly improve the credibility of the results uncovered by a tool in bio-medical analyses.

It would be very beneficial for researchers to have a simple web interface that provides only the most important options, yet still incorporates the full power of a network analysis tool. This would enable more researchers to use the tool while improving accessibility and user-friendliness.

Large amounts of experimental data are evaluated using different scripting languages and tools, such as the R statistical framework [21]. Due to the diversity of developers, many preferences regarding implementation platform and framework exist. Since developers often do not have the resources to support several frameworks, the usability of said tools is limited, and provides the need for seamless integration between enrichment tools and other platforms.

If such a tool was available as a web service, however, and using a commonly known interface, such as JSON (JavaScript Object Notation) for communication, it would have several advantages. Not only would the developers only need to support one core platform, but the researchers using the service would move the calculations to a dedicated powerful server, giving them results faster. Such a seamless integration with an enrichment tool would lessen the need for manual transfer between systems, and thereby increase the research effectiveness. Furthermore, providing a tool not only as a standalone application, but also as a web service could help to overcome gaps between programming/scripting languages.

1.2 Aim

KeyPathwayMiner [2] is a network enrichment tool that is available as a Cytoscape app and as a standalone command line tool. The aim of this thesis is to improve KPM, such that it overcomes the aforementioned issues. This will be done by firstly analyzing the state of the art in the field. Several network enrichment tools will be compared with regards to the features/requirements identified. Subsequently, new features of KPM introduced in this thesis will be explained, followed by implementation details. A thorough description will follow of how KPM was modularized and adapted to allow support for a stand-alone version, a Cytoscape app, as well as a newly developed web application. All additions made to KPM will be tested, and followed by a discussion of the results in the light of the identified requirements. Lastly, a summary of the project as a whole will be provided, as well as an outlook to future work in the area.

2 Requirements analysis

In the era of big data, extracting knowledge from vast amounts of experimental data in a meaningful way becomes more and more important. Several tools have been developed in the past to address this issue. In the following section, we will shed a light on the requirements of data mining tools with focus on data mining from biological networks. We will refer to these tools as network enrichment tools.

2.1 Deployment

Researchers work on a variety of different platforms, including Windows, Linux and MacOS. This leads to the basic requirement of platform independence (R1).

Complex network analysis benefits from a software platform that allows data to be analyzed and visualized in a user-friendly way. For biologists, one of the most popular tools is Cytoscape [22], which is an open source bioinformatics software platform. It is used to visualize interaction networks, where it is possible to integrate additional data, such as gene expression profiles. It contains functionality for producing network statistics, and downloading networks and gene notations from publicly available databases, such as the projects Gene Ontology [13] and KEGG [18]. It is therefore advantageous if a network enrichment tool can be used as a Cytoscape app (R2), facilitating direct and easy access to existing networks. In addition, this will allow biologists without programming skills to integrate different approaches in the form of various apps.

However, while Cytoscape is a convenient and user friendly tool, it is also contains a complex user interface and installation process which might already be a barrier for some users. Additionally, if users do not need to visualize their datasets it should be possible for users to run a tool even without having to install additional software (R3).

As the popularity of network visualization tools change, analysis tools should be able to follow the shift.

Migrating from one programming platform to another can be a complex task, especially when it comes to re-engineering the core algorithms, e.g. ensuring they contain the same functionality and provide the same results. To overcome such issues, a tool should support integration (R4) into other systems. This will minimize maintenance, and optimize both dependability and stability. If an arbitrary tool should integrate with another, a common communication interface is needed. if one tool can be included as a sub-part of the other it is called modularization. If this is not possible, then communication needs to be performed through a common service based communication interface. One popular example are the so-called Representational State Transfer based web services, also called RESTful web services. They use the HTTP protocol to allow lightweight and scalable communication through the JSON or XML standards.

While all researchers have access to computers, not all of the workstations are designed to perform complex computations on network graphs. For heavy tasks like this, a centralized web service is preferable, and it should contain a minimalistic and easily accessible user interface, such that the user is not challenged with overly complex menus. A centralized web service could also entail the possibility of saving results remotely, such that no invaluable research is lost if a workstation is damaged or lost. Most importantly, with a centralized web service (R5) there is no need to deploy the tool on every workstation, making it easier to ensure everyone is using the most recent version of the tool. Additionally, while a tool running on a local workstation might take hours to complete, using a dedicated server for the job might cut this time down to minutes, as all calculations would be performed on a dedicated server created for the tool.

2.2 Network perturbation

When performing enrichment tests, knowing if your results are meaningful, or just the product of chance, is a reoccurring problem. To overcome this problem, one can try to alter the network, or dataset, to a certain degree, and compare this with the original result. The amount of overlap between the results is a function of the robustness of the technique used to get them, e.g. the higher the overlap, the more robust the technique. Network perturbation (R6) as a part for robustness and validation is crucial for assessing quality of results through complex analyses. Currently, this is missing in all popular tools, even though it can provide the user with new information about the quality of the results they have found. In this context, robustness analysis is when the multiple results are compared to the original run with same settings, only where the graph is perturbed to a certain degree. When generating multiple connected components in a network, there is a chance that there is an overlap of one or more nodes between two components. Therefore it would be a useful feature to include removal of such Border Exception Nodes (BENs), ensuring they are only counted once towards the amount of exception nodes.

2.3 Integrate multiple datasets

It is of increasing interest to analyze multiple OMICS datasets at the same time, connecting them in various ways to uncover new knowledge from the results. Gene expression data alone has brought many insights, but only when several layers of data are integrated can we unravel the complex mechanisms underlying the biological systems, since these are also regulated and influenced on multiple layers. This means that there is a need for integrating multiple datasets (R7) into the analysis.

2.4 Visualization of summary statistics

Most enrichment tools provide some sort of specialized statistics based on the results from the technique used, for example the amount of exception nodes found by running KPM. In the current state of the studied tools, all are still missing an integrated visualization of these results (R8) in the form of informative visualization, network graphs, box-plots, etc. Such functionality would enable the researcher to decide more efficiently whether the test has uncovered useful results.

3 Enrichment tools

Several projects have been developed for network enrichment. For the following section we have compared a selection of related tools (Table 2). For a comparison of all relevant tools see the appendix on page 55. We illustrate to what extent each of the projects fulfill the requirements defined in Section 2.

While there are many projects that are used in the field of network enrichment, which is sometimes also referred to as active module detection, only the most popular tools will be covered. It should be mentioned that the focus here is on the properties of the tools and not on the various underlying algorithms for network enrichment.

3.1 jActiveModules

The jActiveModules project [15] was the first program to implement the idea of extracting affected pathways from repositories of interaction data. Gene expression profiles are used to identify and extract differentially expressed sub-networks when given a biological network. It is currently only available as a Cytoscape plugin in Java.

3.2 MATISSE

The MATISSE[1] (Module Analysis via Topology of Interactions and Similarity SEts) project [23] uses efficient computational methods for detecting and analyzing so-called Jointly ACtive Connected Sub-networks (JACSs). It is implemented as a Java application, but only supports Windows® as a platform. Currently there is no way to incorporate multiple datasets, or perform network perturbations.

3.3 HotNet

The HotNet project [25] works by using a diffusion process on the network to define local neighborhoods of influence. Then a two-stage multiple hypothesis test is derived, bounding the false discovery rate associated with the identified sub-networks. HotNet is written in Python and is therefore platform independent by design, if no platform specific libraries are used. It differs from most of the other projects investigated here, as it was one of the only projects providing web access.

3.4 KPM

KeyPathwayMiner 4.0 [4], is a network enrichment tool written in Java. It was first released in 2011 as a plugin for Cytoscape, and has since went through several iterations of improvements, improving usability and features at each iteration. The primary addition in [4] was the extension to OMICS data integration, such that the user is now able to combine multiple OMICS data types. There has been included a feature for search bias to the sub-networks, such that it is now possible to define positive and negative nodes. This means the sub-network is now forced to contain the positive nodes, and avoid the negative nodes.

3.5 Summary

It is evident from the requirements table that there are gaps between the available tools, as they are, and the uncovered needs of the researchers. In particular, there was a complete

[1] `http://acgt.cs.tau.ac.il/matisse`

lack of support of perturbation of networks and datasets. Additionally, though all included some form of visualization, none of them integrated a visualization of statistical results.

Requirements	
Platform independent	R1
Cytoscape	R2
Standalone	R3
Support integration	R4
Web access	R5
Network perturbation	R6
Multiple datasets	R7
Visualization of summary statistics	R8

Table 1: This table listing the requirements identified as important for state of the art network enrichment tools.

	(R1) Platform independence	(R2) Cytoscape	(R3) Standalone	(R4) Support integration	(R5) Web access	(R6) Network perturbation	(R7) Multiple datasets	(R8) Visualization of sum. stats.
HotNet	✓	✗	✓	✓	✓	✗	✗	✗
MATISSE	✓	✗	✓	✗	✗	✗	✗	✗
jActiveModules	✓	✓	✗	✗	✗	✗	✗	✗
KPM 4.0	✓	✓	✓	✗	✗	✗	✓	✗

Table 2: Requirements table, describing how each enrichment tool fulfill the uncovered requirements. Full table in Appendix A.

3.6 Outline

While KPM 4.0 is a powerful enrichment tool, it is still missing features, such as web access and network perturbation. In this work, KPM will be extended to cover all of the identified missing features. More specifically, a website (R5), along with a web service (R4), such that the users can freely use the website to perform their calculations, or integrate the service into their own application. The web service will use the JSON format, which is among the most common interfaces for inter platform communication. Several algorithms for perturbation of networks (R6) will also be implemented into the core of KPM, with its functionality visible in both the Cytoscape plugin and the website/service. Additionally, methods for creating statistical charts, e.g. box-plots, will be implemented to visualize the results (R8) of the enrichment algorithm runs. Lastly, there will be a discussion on the usability and state of all the implemented features, followed by an outlook on what could be next for KPM.

4 Material

In this section an overview of different material and technologies used in the course of this dissertation will be provided. This section will end in a description of KPM in the state it was before the work for this thesis began, providing a bridge to the methods used as a part of the work performed.

4.1 Groovy/Grails

Groovy is a new programming language for the Java Virtual Machine.The syntax for Groovy resembles a script-like version of Java, with a number of simplifications. Groovy also supports the use of pure Java code and direct integration with pure Java JARs. After its official release in 2007, several frameworks have been implemented using groovy as a basis. One example is Grails[2], which is an open source web application framework based on a "Model, View, Control"-architectural (MVC) pattern. It is used in this project for solving part of the deployment requirements from Section 2. In addition to MVC, Grails natively implements Hibernate framework, which is a Object/Relational Mapping (ORM) framework, for persisting data without having to write any SQL.

4.2 JFreeChart

JFreeChart [14] is a free Java chart library, which provides an extensible API for creating and displaying charts in a Java application. It also supports export of charts to popular formats, such as PDF and PNG.

4.3 R Statistical Framework

R [21] is a language and environment for statistical computing and graphics. It contains a wide variety of statistical techniques, and is highly extensible. The popularity of the R framework is primarily due to its rich plugin ecosystem, where many researchers contribute. In this project, mainly two R packages are used. The first one is called Curl [12], which provide an API for creating and consuming services, whether it be local or web services. The other package is called RJSON [9] which is which provide functionality for converting R objects to JSON format and back. In this project, R is used as an example for an application case of inter-platform use of KPM.

4.4 Network visualization in a web application

When selecting a technology for displaying large network graphs, several factors have to be taken into account. Firstly, it should support visualization in a browser, and optimally show information about a node when hovered. Functionality to download the graph as an image in a acceptable quality is also preferred. To maximize extendability, the graph definition should be on the JSON format, such that export is simplified.

One such technology is the JavaScript library called Sigma [16]. This library takes a JSON encoded graph as input, and its API supports changing size, color and labels of both nodes and edges. Sigma uses the HTML element `Canvas` to render the graph, making the networks scalable and simple to export to image formats such as PNG. This library was chosen because of its extensive and intuitive API, over similar mature libraries like D3 [6] that are poorly documented.

[2]`https://grails.org/`

4.5 Cytoscape

Cytoscape [22] is an open source desktop application written in Java. It is used as a platform for visualizing biological pathways and molecular networks using advanced graph techniques to effectively handle huge datasets and graphs, which makes it ideal for systems biologists. This application is highly extensible, as it is possible to include your own code as plugin, called an "app". As a part of this thesis, extensions were made to an existing app written for Cytoscape.

4.6 KeyPathwayMiner (KPM)

KeyPathwayMiner operates on a network $G = (V, E)$ with vertices or nodes V and edges E, as well as a binary indicator matrix M. The columns of this matrix typically correspond to different samples or patients, whereas the rows of the column correspond to the nodes V. For gene expression, such a matrix can be defined as

$$M(g,p) = \begin{cases} 1 \text{ if gene g is differentially expressed in patient } p \\ 0 \text{ if not} \end{cases} \tag{1}$$

In other words, a 1 in this matrix indicates that a given gene $g \in V$ is active in a patient or sample p. The purpose of KPM is to search for all connected maximal subgraphs $G' \in G$, where all but k nodes of the subgraph G' are active in all but at most l cases. To solve this computationally hard problem, different search algorithms and strategies have been implemented [3]. The following is a brief description of how they work.

4.6.1 Greedy - Algorithm

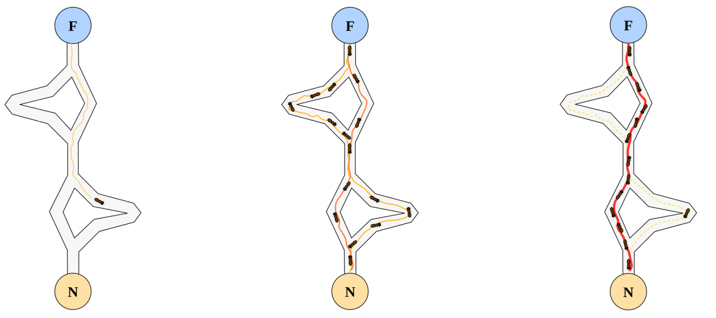
The greedy approach for solving this problem does not ensure an optimal solution, but yields a set of solutions. It works by looking at the set of all exceptions from the l sub graph found from the main graph. For each vertex u it constructs a new set $W = u$. A new vertex v is added to the set W iteratively. v is required to be adjacent to W, and to maximize the number of nodes connected to W without passing through an exception node that is not a part of W, and we call this set S, e.g. we seek to maximize $|S(W_u) \cup \{v\}|$. The iterations stops when $|W_u| = k$, and the $S(W_u)$ of maximal size is returned.

4.6.2 Exact Fixed Parameter Tractable (FPT) - Algorithm

The exact algorithm is based on a branch and bound approach, which works by creating a search tree of partial solutions. First, a fitness value has to be defined, which allows to establish a lower bound for eliminating sub-optimal solutions that can be removed from the search space, as well as an upper bound that represents the optimal solution. In KPM, this algorithm is applied by having a lower bound as the best current solution and an upper bound computed as follows. Given a number x, we look at a partial solution already containing $k-x$ exception nodes that can be reached from the current subgraph. The algorithm will then find all exception nodes reachable within x steps. The x nodes with the highest weights are taken, and added together. This fitness value provides the upper bound for that partial solution. In the end, the solution with the highest fitness value is returned and guaranteed to be optimal.

4.6.3 ACO - Algorithm

An improved heuristic Ant Colony Optimization [10] approach was also implemented as a part of KPM. Algorithms in the Ant Colony Optimization (ACO) family are probabilistic methods, used for solving problems which can be reduced to finding paths in network graphs.



(a) The first ant finds a food source F. Once it starts returning, along a path to the colony N, it releases a pheromone that fades over distance.

(b) The ants traverse all the possible paths, from the source to the colony, and a higher pheromone level means that an ant is more likely to take that path.

(c) The ants will follow the shortest path, as the pheromone level will be the highest. The pheromone level of other paths will fade away eventually.

Figure 1: Visualization of how ACO algorithms are inspired by how ants find the shortest path to their destination. Image taken from [11].

4.6.4 Search strategies

As mentioned before, the main objective of KPM is to extract all maximal connected subgraphs, in which all but exactly k nodes are active in all but at most l cases [5]. This method was coined INES (individual node exceptions) and had some bias towards favoring hub nodes. Therefore, another approach called GLONE (global node exceptions) was developed. In the GLONE algorithm, the objective is very similar. The difference is that the nodes are required to be inactive in at most l cases in total, distributing the exceptions across all nodes in a solution [5]. It could be proven that GLONE can solve the problem with a worst case run time of $O(|V|^k + |V|^3 + |E|)$ [2].

4.6.5 Network data

KPM works with network graphs, either converted from the Cytoscape native format, or the Simple Interaction Format (SIF). The SIF file format consists of a list of edge definitions (Listing 1). It defines and edge by first describing the node id of the first endpoint, continued by a name of the interaction type, and ends with the node id of the second endpoint.

Listing 1: **SIF file format**

```
<nodeid> <interaction-type> <other-nodeid>
23162    pp    4214
```

```
2033    pp    7251
573     pp    4233
156     pp    9590
```

The graph used to test the additions of this thesis, is the protein-protein interaction network from Ulitsky et al 2008 [24], called `graph-ulitsky-entrez.sif`. This network contains 7,384 nodes and 23,462 edges, which is a typical size of a PPI network, and shows the magnitude of the graph KPM normally handles.

4.6.6 Datasets

In KPM, The format of datasets used for the enrichment analysis is a matrix stored in tab-separated format. The first column describes node ids. Nodes can be genes or proteins, for instance. The remaining columns represent samples in which a node can be active (1) or inactive (0). How the state of each node in a sample is determined is up to the user and needs to be deal with during pre-processing. In gene expression data, for instance, this could be interpreted as the question if a gene is differentially expressed in a sample or not. It is important to note that KPM can only handle discrete matrices like this, also referred to as indicator matrices, but not matrices with continuous values.

Listing 2: **Dataset file format**

```
80790    1  1  1  0  0  ...
2903     0  0  0  0  0  ...
90850    0  0  0  0  0  ...
645937   0  0  1  0  0  ...
4281     0  0  1  0  0  ...
```

The dataset used for testing in the thesis is called `COAD-EXP-UP-p0.05.txt`, which originate from the National Human Genome Research Institute's Cancer Genome Atlas[3], as a part of their research into colon cancer [8]. This specific dataset is from a colon adenocarcinoma sample, which contains information about 27,766 genes 625 tumor samples.

[3]`http://cancergenome.nih.gov`

5 Methods

Significant progress has been made in KPM 4.0, as shown on the requirements table (Table 2). However, even though KPM already produces results that were shown to have biological relevance [4], users would benefit from assessing the reliability of results automatically in the form of robustness analyses and network perturbations.

5.1 BEN removal

A challenge was mentioned in Section 2.2, which was the importance of ensuring that we only count exception nodes once, as it would otherwise give an erroneous perspective of the results if counted multiple times. This merits the following algorithm, which solves exactly that. The algorithm uses a conversion from a `Result` containing a list of visited nodes to a sparse graph. In this structure a node has a variable called `isValid`, and if `false` is seen as an exception node. The following algorithm is executed after this conversion has happened.

First we start with a list of nodes to check, which at start holds the entire list of exception nodes. Then we loop, while there are still exception nodes left, and look at the next node from the list. This node, along with the incident edges, is removed from the graph. The connected components from that node is found. The connected components are found using a breadth-first search which finds all nodes connected to a root node. Afterwards, we check the amount of valid connected components, and if a component consists only of exception nodes, all the nodes contained within are removed from the graph. Otherwise, a counter for valid connected components is incremented. Once this check has finished, and if the amount of valid connected components is greater than or equal to two, we re-add the node, and its incident edges, from the graph. Once there are no more nodes to check, the loop finishes and the graph, without the border exception nodes, is returned.

Listing 3: **Algorithm 1 - removeBENs(SparseGraph graph)**

```
nodesToCheck = at start: the list of exception nodes

while nodesToCheck.size > 0 do
   nextNode = next node from nodesToCheck
   remove nextNode and incident edges from graph
   ccs = connected components found through BFS search
   for each cc in ccs do
      validCC = starts at 0, amount of valid CCs.
      if cc.nodes only consist of exception nodes
         remove all cc.nodes from nodesToCheck
         remove all cc.nodes from graph
      else
         validCC++
      end if
   end for
   if validCC >= 2 then
      re-add the nextNode from graph
   end if
end while

return graph
```

It is worth noting that the code for finding connected components utilized a feature from the JUNG [26] library. This library was already used as a basis for the working graph in KPM. The feature is called `BFSDistanceLabeler`, and as the name implies, it labels the distance from one node to all other nodes in a graph. Once this is done for a node, all the nodes connected to this node is removed from the list of nodes to visit, and a new source for the BFS labeler is decided, and the process starts over. In the end all connected components are found and returned and used in the code above. In the end, this algorithm returns the input graph minus the BENs.

5.2 Robustness and validation analysis

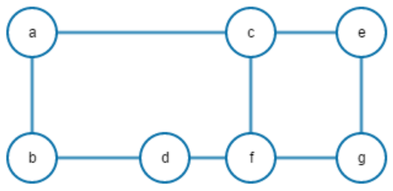
The importance of performing robustness and validation analyses was emphasized in section 2.2, and as part of this thesis, several algorithms for perturbing a network graph has been implemented. The important aspects of each of the algorithms will be explained, which will then lead to a description of how these algorithms were applied in KPM.

The following perturbation algorithms all have the same input parameters. The input parameter `percentageToPerturb` should always be between 0 and 100, since it represents the percentage of the total amount of elements that needs to be perturbed. As well as the KPM network graph on which the perturbation should be carried out. Common to the perturbation algorithms is also the random number generator, which should be assumed to always return a number between 1 and the length of either the total amount of edges or nodes, depending on the algorithm.

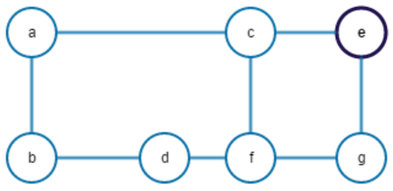
5.2.1 Node remove perturbation

This algorithm has the purpose to remove a certain percentage of the nodes from the input graph, with no regards to preserving the properties of it.

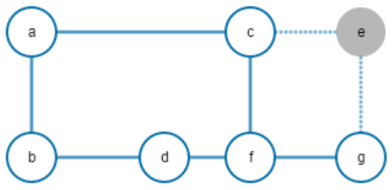
The algorithm starts by finding all nodes in the graph, and then calculating how many nodes the input percentage corresponds to. Then we use a random number generator, which generates numbers between 1 and the length of the list of all non-removed nodes. This is done through a method called `getNextRandomIndex()`, which handles the interaction with the random number generator, and the `indexList`. Next, we loop until we have removed as many nodes as correspond to the given input percentage. A node is found from the integer gotten using the result of executing `getNextRandomIndex()`, as the index of the node to remove. Then we remove the node from the graph, add it to the list of removed nodes and decrement the remaining number of removals needed. Once the loop has completed, the graph, with nodes removed, is returned.



(a) The graph before the algorithm



(b) First step of the perturbation is to find a random node, here marked with a black border.



(c) Since node e has not already been removed, we mark it as removed and delete it from the graph. We return the graph consisting of a, b, c, d, f and g.

Figure 2: Visual explanation of the loop-part of the node remove algorithm.

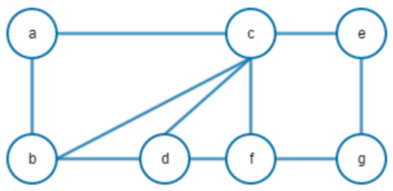
Listing 4: **Algorithm 2 - nodeRemove(int percentageToPerturb, KPMGraph graph)**

```
removedNodes = List of nodes already removed, starts empty
nodes = array of all nodes in the graph
 indexList = List of all nodes-list-indexes that are yet to be drawn.

nrNodesToRemove = ceil( (nodes.length / 100) * percentageToPerturb )

while nrNodesToRemove > 0 do
   node = nodes[ indexList[getNextRandomIndex()] ]

   remove node from graph
   add node to removedNodes
   nrNodesToRemove--
end while

return graph
```

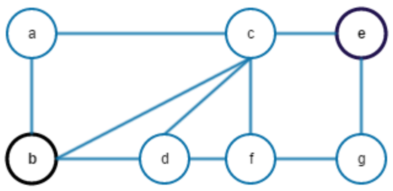
5.2.2 Node swap perturbation

This algorithm simply swaps the labels on edges in the graph, with no regards to the properties of the graph.

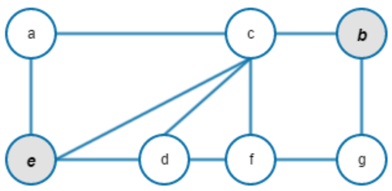
This algorithm, like the "Node remove"-algorithm, starts by converting the input percentage to the total amount of nodes to be swapped. The same `getNextRandomIndex()`-method is used to draw random unused node indexes. Two nodes are found this way, and if the nodes are not already swapped and not the same node, we go through all edges of the graph and swap the labels for one with the other, and vice versa. Then we add the two nodes to the list of swapped nodes and lower the number of nodes left to swap by two. When no more nodes need to be swapped, the loop ends and we return the graph with node labels swapped.



(a) The graph before the algorithm.



(b) Using `getNextRandomIndex()`, we find two random nodes, *b* and *e*, that have not already been swapped.



(c) Once all the edges have been updated, and it looks like the two nodes, *b* and *e*, have switched place, and are marked as used.

Figure 3: Visual explanation of the loop-part of the node swap algorithm.

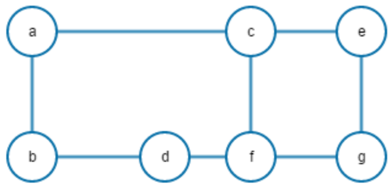
Listing 5: **Algorithm 3 - nodeSwap(int percentageToPerturb, KPMGraph graph)**

```
swappedNodes = list of nodes already swapped, starts empty
nodes = array of all nodes in the graph

nrNodesToSwap = ceil( (nodes.length / 100) * percentageToPerturb )

while nrNodesToSwap > 1 do
   node1 = nodes[ indexList[getNextRandomIndex()] ]
   node2 = nodes[ indexList[getNextRandomIndex()] ]

   if node1 == node2
      or swappedNodes.contains(node1)
      or swappedNodes.contains(node2) then
      continue
   end if

   for all edges in graph do
      swap labels for node1 with node2 and vice versa
   end for

   add node1 and node2 to swappedNodes
   nrNodesToSwap = nrNodesToSwap - 2
end while

return graph
```

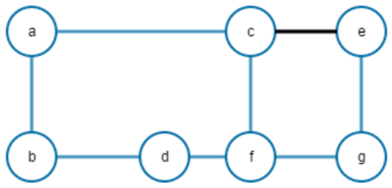
5.2.3 Edge remove perturbation

The algorithm is similar to the "Node remove"-algorithm, Section 5.2.1, but instead of nodes, edges are removed.

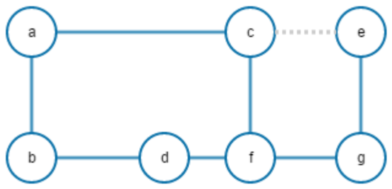
This algorithm start by converting the percentage to perturb into the actual amount of edges to remove, and we start a loop that runs while there are still edges that needs to be removed. Then we get a random edge, using `getNextRandomIndex()`, and only remove the edge if it has not already been removed. Afterwards, we remove the edge from the graph and adds it to the list of removed edges, and decrement the remaining amount of edges that needs to be removed. When we have removed enough edges, the loop ends, and the graph is returned.



(a) The graph before the algorithm.



(b) Using `getNextRandomIndex()`, we find a random edge.



(c) We mark the chosen edge as used, and removes it from the graph. Once enough edges have been removed, we return the graph without the removed edges.

Figure 4: Visual explanation of the loop-part of the node swap algorithm.

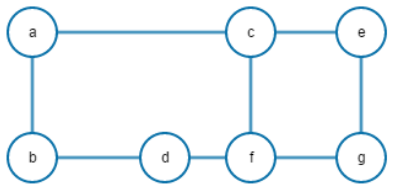
Listing 6: **Algorithm 4 - edgeRemove(int percentageToPerturb, KPMGraph graph)**

```
removedEdges = List of edges already removed, starts empty
edges = array of all edges in the graph

nrEdgesToRemove = ceil( (edges.length / 100) * percentageToPerturb )

while nrEdgesToRemove > 0 do
   edge = edges[ indexList[getNextRandomIndex()] ]

   if removedEdges contain edge then
      continue
   end if

   remove edge from graph
   add edge to removedEdges
   nrEdgesToRemove--
end while

return graph
```

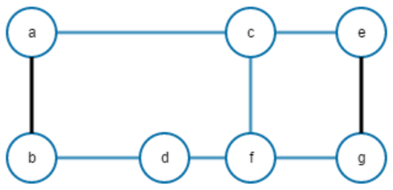
5.2.4 Edge Rewire perturbation

This edge rewire algorithm was published in Maslov et al [19]. It preserves the node degree, as well as the number of immediate neighbors.

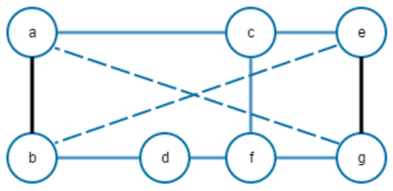
We start by calculating how many edges correspond to the input percentage. We then loop until enough of the edges have been rewired. Inside the loop we get two random edges, consisting of two vertices, (v1, v2) and (u1, u2). Afterwards we check whether the edges share at least one vertex or if one of the edges have already been rewired. If so, we restart the loop, otherwise we create two new edges, (v1, u2) and (v2, u1), add them to the graph, and remove the old ones, (v1, v2) and (u1, u2). Lastly we lower the remaining amount of edges that needs rewiring by two. Once the loop has ended, we return the graph, with its edges rewired to the desired degree.



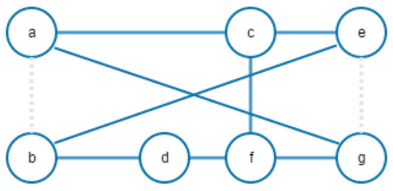
(a) The graph before the algorithm.



(b) Using `getNextRandomIndex()`, we find a two random edges, (a, b) and (e, g). Since we they have no vertices in common, we can do the next step.



(c) Then we add the edges (a, g) and (b, e), and mark the original edges, (a, b) and (e, g) as deleted.



(d) The rewiring is complete and the updated graph can be returned.

Figure 5: Visual explanation of the loop-part of the edge rewire algorithm.

Listing 7: **Algorithm 4 - edgeRewire(int percentageToPerturb, KPMGraph graph)**

```
rewiredEdges = list of edges already removed, starts empty
edges = array of all edges in the graph

nrEdgesToRewire = ceil( (edges.length / 100) * percentageToPerturb )

while nrEdgesToRewire > 1 do
   edge1 = (v1, v2) = edges[ indexList[getNextRandomIndex()] ]
   edge2 = (u1, u2) = edges[ indexList[getNextRandomIndex()]]

   if (v1 == u1) or (v1 == u2) or (v2 == u1) or (v2 == u2) then
      continue
   end if

   if wiringExists(edge1) or wiringExists(edge2) then
      continue
   end if
```

```
    create edges (v1, u2) and (v2, u1) in graph
     delete edges (v1, v2), and (u1, u2) from graph

    nrEdgesToRewire = nrEdgesToRewire - 2
end while

return graph
```

5.2.5 Matrix shuffle perturbation

In addition to perturbing graph input, it would be advantageous to enable perturbation of the input dataset files. The input and output will be a string representation of the dataset. The following algorithm (Algorithm 8) performs column wise perturbation, in which the parameter defined percentage perturbation refers to the amount of columns, in which all of the 1's will be shuffled. The condition for one such column to be included, is that it must contain at least one cell with a value of one. By using the same randomization scheme for finding both the columns and the cells to change, it is not possible to write in the same cell twice. It is known that cases exist, in sparse data matrices, where it is not possible to shuffle enough columns, since there are not enough columns with values.

Listing 8: **Algorithm 5 - columnShuffle(int percentageToPerturb, String dataset)**

```
datasetMatrix = dataset, converted into a matrix[][]

nrColumnsToShuffle = ceil( (datasetMatrix[0].length / 100) * percentageToPerturb )

while nrColumnsToShuffle > 0 do
    columnIndex = indexList[getNextRandomIndex()]

    column = datasetMatrix[ columnIndex ]

    amountOnes = amount of 1s in column

    if amountOnes == 0 do
        continue
    end if

    for i from 1 to column.length do
        datasetMatrix[columnIndex][i] = 0
    end for

    for i from 1 to amountOnes do // assume inclusive nr
        datasetMatrix[columnIndex][ cellIndexList[getNextRandomIndexForCell()] ] = 1
    end for

    nrColumnsToShuffle = nrColumnsToShuffle - 1
end while

newDataset = datasetMatrix converted back into a String

return newDataset
```

5.2.6 KPM with perturbation

Before this thesis, it was possible to execute KPM either as single run or as a batch run with varying node exception and case exceptions. A contribution from this thesis is the integration of the perturbation algorithms from the previous subsections. The integration works by adding a layer on top of the existing code that executes KPM. This layer is only executed if it is explicitly chosen as it is an object for itself, as will be explained in a later chapter.

The workflow, shown on Figure 6, displays what happens when a user selects one of the three run types. If it is the normal run type, no further input is needed from the user, and the run is ready. If either of the other two run types are selected, then selection of the perturbation type is needed, as well as setting up the perturbation values.

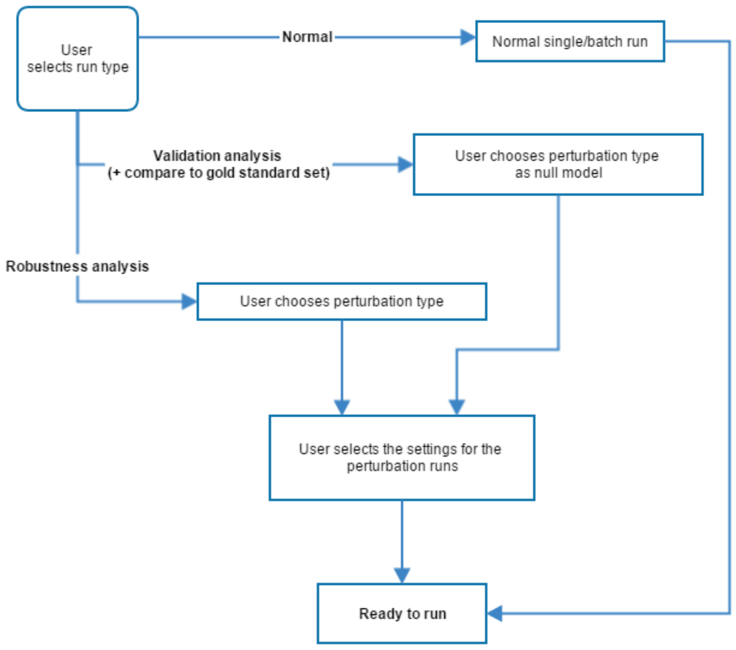

User selects run type
Normal
Normal single/batch run
Validation analysis (+ compare to gold standard set)
User chooses perturbation type as null model
Robustness analysis
User chooses perturbation type
User selects the settings for the perturbation runs
Ready to run

Figure 6: The perturbation options change whether it is a validation- or robustness-analysis. For the validation analysis, the options are "degree preserving rewiring" and "Node label perturbation". For robustness analysis they are "edge-removal" and "node-removal". The perturbation settings are start, step, and max percentage value, and amount of graphs per step, for the perturbation runs.

After the setup is complete, the rest is automated. KPM will then first do a run with 0% perturbation, henceforth called the original run. Afterwards, the perturbation runs are performed by iterating through all the percentage perturbations that was setup, and number of repetitions that were chosen during the setup. (Figure 6).

5.2.7 Jaccard coefficient

An overlap comparison between the original run and the perturbation runs is performed, once the perturbation runs has completed. Instead of just providing the absolute overlap, we calculate the Jaccard Coefficient (Eq. 2), which will provide us with a more informative image of the results. The Jaccard coefficient measures the similarities between two sets A and B, and returns a value that goes towards 1 the more similar they are.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

6 Implementation

When the work in this project began, the Java source code for KPM was scattered between multiple projects, two for Cytoscape and one for a standalone version. Since it was known that a new Java website/service was scheduled for implementation, best practice dictates to minimize the amount of code that needs to be maintained, and optimize the reusability. To this end, it was decided to extract the core functionality of KPM into a single independent Java module, in order to optimize reusability.

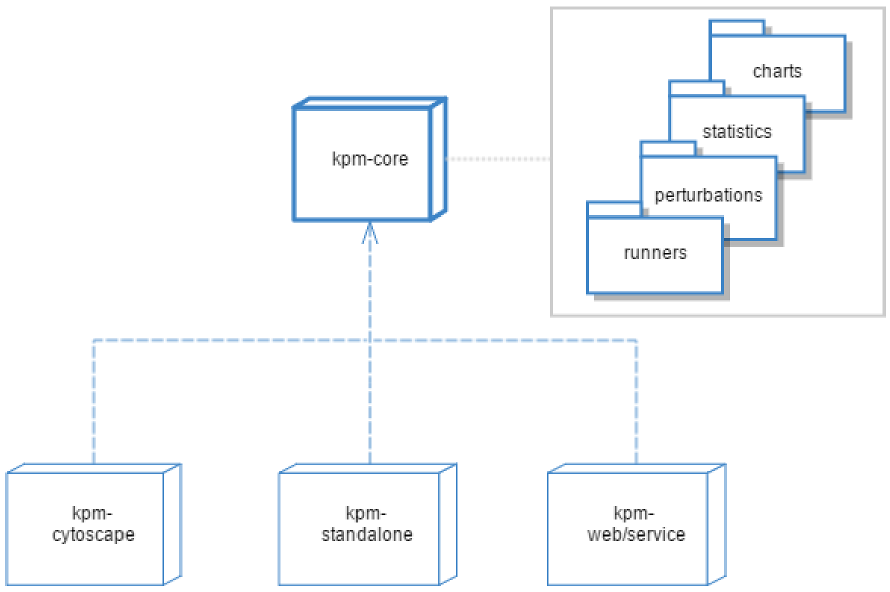


Figure 7: Diagram of how the different KPM implementations, after this project, all depend on a single core KPM module. The kpm-core module includes all the features that should be shared among all platforms, including charting and statistics. With this approach, it is possible to keep a high level of maintainability for the core features of KPM. Each of the packages included in kpm-core is explained in Section 6.1.

In the Figure 7 it is shown how KPM is no longer a set of scattered projects, with a different implementation in each module, but rather a single core module, with an interface module for each platform it should support. Since this extraction were known from the beginning of the project, some important factors was also taken into account:

- *Charts should be available on all platforms*:
  The KPM project will spans across almost all common platform: command-line, application, website, web service. Even with this wide variety of platforms, the generated charts should still look the same. This will give the user a feeling of familiarity, no matter what platform is chosen. The detailed explanation of how this was solved, is described in Section 6.1.3.

- *Monitoring the progress of a run*:
  KPM did, in its original form as a Cytoscape plugin, support a so-called `taskmonitor`. The `taskmonitor` is a class from the Cytoscape library, used to broadcast updates from the algorithm, notifying the user of the progress. This class, however, is not supported in projects that are not Cytoscape-plugins, as they would need to include references to the Cytoscape-framework. Thus, a generalized interface is needed for communicating

between the KPM run and the user, which can used across all interfaces. The design and implementation of how this need was satisfied will be explained in Section 6.1.5.

- *Threading across multiple environments*:
  Even though Java is known for being supported on most platforms, the way each platform supports threading differs for each of them. There is a particular difference in the way websites handles threads for a user, in comparison to a normal application.

  Because all interaction with the service is performed through posts and request to a service, there has to be added additional features, which will be explained in Section 6.4.1.

The following subsections will explain, in detail, the concepts of the different modules, along with an description of their most important features. First, an explanation of the `kpm-core` project, which holds the actual algorithms and important features.

6.1 KPM-Core

The previous section described points that should be taken into account during the development of this module. In addition to showing how each point is satisfied, the implementation and usage of the perturbation algorithms from Sections 5.2.1-5.2.4 will be explained.

The implementation of the `kpm-core` Java/Maven module partly builds on the concept that if objects have several methods or properties in common, they should inherit from the same base class. While this is not true in all cases, it does hold for objects that are contextually very similar, such as perturbation types with methods based on a graph and a percentage. Additionally, the design pattern Facade has also been used. It enforces the use of interfaces to minimize the impact of changes to code have for the consuming objects. Since the usage-pattern of `kpm-core` means we have multiple consumers, this pattern can contribute to higher maintainability. These approaches are most visible in the implementation of perturbations and charts, as shown in Figure 8.

The `kpm-core`-module has implemented five main packages (Figure 8). The `runners`-package contain the classes used for actually running the KPM algorithms, and are wrappers for accessing them, starting runs, et cetera. The `perturbation`- and `charts`-package contains the implementations of the perturbation algorithms and charts, respectively. They both also implementing inheritance and facade patterns. The `statistics`-package contains classes for performing the statistics used as part of the results during the runs. The `taskmonitors`-package includes an interface for communicating between the runners and the user.

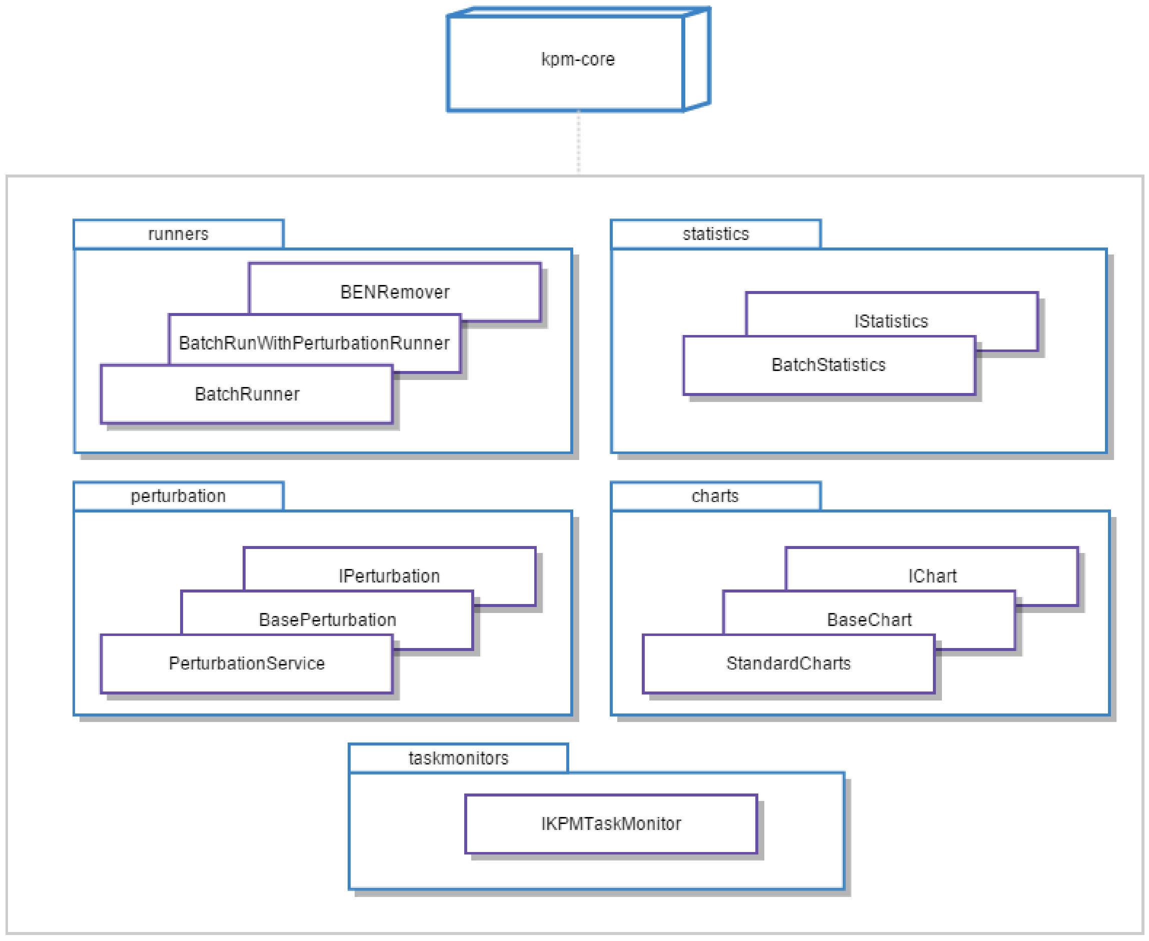


Figure 8: A visualization of how the `kpm-core` module features are build up. Here is only shown the parts implemented as a part of this dissertation.

6.1.1 Perturbations

Because of the Facade pattern, all perturbation classes implement the `IPerturbation`-interface (Listing 9). This interface defines the structure of a perturbation class, including the execute method which is where the perturbation actually takes place. This interface is first implemented by the `BasePerturbation`-class, and the four algorithms were implemented as the classes:

`NodeRemovePerturbation` (Section 5.2.1)

`NodeSwapPerturbation` (Section 5.2.2)

`EdgeRemovePerturbation` (Section 5.2.3)

`EdgeRewirePerturbation` (Section 5.2.4)

Listing 9: **Interface - IPerturbation**

```
public interface IPerturbation {
    public KPMGraph execute(..., IKPMTaskmonitor taskmonitor);
    public PerturbationTags getTag();
    ...
    public enum PerturbationTags{ NodeSwap, EdgeRemoval, NodeRemoval, EdgeRewire };
}
```

Instead of having a single class containing all perturbation algorithms, it was decided to have a class for each type of perturbation. This makes maintenance easier. The algorithms themselves are implemented in each class' `execute`-method. A task monitor, implementing the `IKPMTaskmonitor`-interface, is given as parameter. The concept of task monitors will be explained in Section 6.1.5. This extra parameter provides additional feedback to the user, or whatever type of consumer is using the permutation class. Note that `IPerturbation` also exposes a method `getTag()`, which provides an identifier enum used by the `PerturbationService`-class to identify the exact perturbation type wanted, without having to know the actual implementation.

The `PerturbationService`-class works as a mediator between the perturbation types and the selection of them. It only implements two methods: `getPerturbation(tag)` and `getPerturbationTags()`. The `getPerturbationTags()`-method returns a list of all the types of perturbations that are available, and `getPerturbation(tag)`, given a valid tag, returns the actual perturbation class, wrapped in the `IPerturbation` interface. This way all implementations of the perturbation algorithms can be private to the core project, and thus satisfying the Facade pattern.

6.1.2 Node overlap comparison

As a part of the validation analysis, it is necessary to compare the results of the original run, with the result of the perturbed runs. In KPM, this is performed by the `NodeOverlapCounter`-class (Listing 10). It works by by counting how many of the gold standard nodes were present in the results. As input, it requires a `KPMSettings` object containing all information and parameters needed for a KPM run. The `compareResultsToGoldStandard()`-method iterates through all the results for a run, and uses the `countOverlap()` to compare the gold standard validation node list to the set of visited nodes, e.g. the result nodes.

Listing 10: **Class - NodeOverlapCounter**

```
public class NodeOverlapCounter {

 public static List<ValidationOverlapResult>
   compareResultsToGoldStandard(KPMSettings kpmSettings)
     { ... }

private static
   List<ValidationOverlapResultItem> countOverlap(
      List<String> goldStandardNodes,
      Map<String, GeneNode> resultNodes)
      { ... }
```

6.1.3 Charts

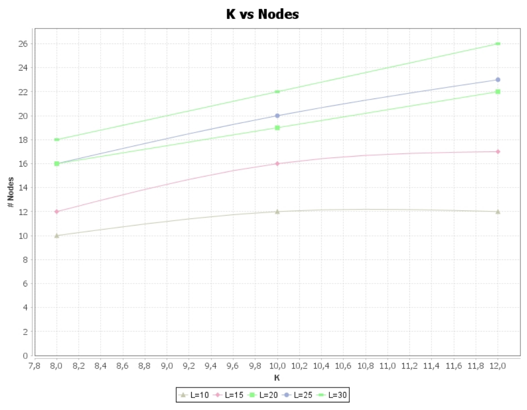
While several charting libraries exist for Java, the most widely used is JFreeChart [14], which was chosen as the starting point in KPM. In this project, four types of charts have been implemented on top of the IChart-interface, all of which implement the `IChart`-interface (Listing 11). By implementing this interface we free the consumers of the `kpm-core` module from referencing JFreeChart. Listing 11 shows how the IChart interface abstracts functionality found in JFreeChart.

Listing 11: **Interface - IChart**

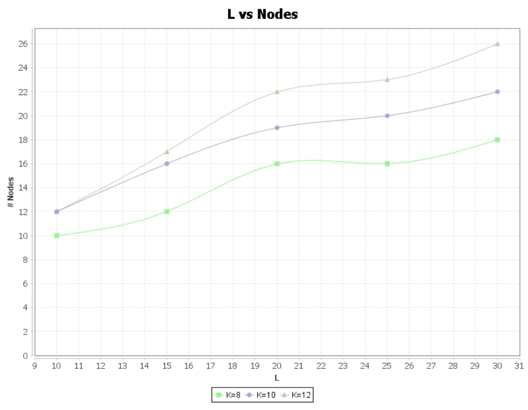
```java
public interface IChart {
   public JPanel getChartPanel();
   ...
   public void saveAsPng(...);
   public void writeChartAsJpeg(...);
}
```

The `getChartPanel()`-method is intended for use with a Java Swing application, where it is possible to show the objects enclosed within a `JPanel`. The method returns the `JFreeChart` wrapped inside a `JPanel`. The `saveAsPng()`-method is for use with a command-line tool, where the chart, on which the method is executed, is saved to the disk as an image in the PNG format. The last last method, `writeChartAsJpeg()`, can be used when a completely in-memory handling of the resulting image of a chart is wanted. In this project, the use case that merits this feature is the web application. Since non-java-applet web applications cannot show JPanels or any other Java entities, the chart has to be converted into a format that can be shown in a browser.

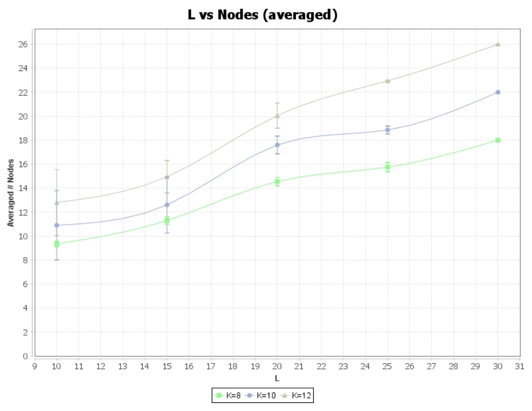
All of the implemented charts, which all implement the `IChart` interface, are `BarChart`, `BoxplotChart` and `XYMultiChart`. The `XYMultiChart` is a line chart. As a part of the standard results, three charts are created (Figure 9), which represent the standard charts that are always needed. These charts are generated through a static class, `StandardCharts`, which take the `KPMSettings` as input.



(a) K vs Nodes



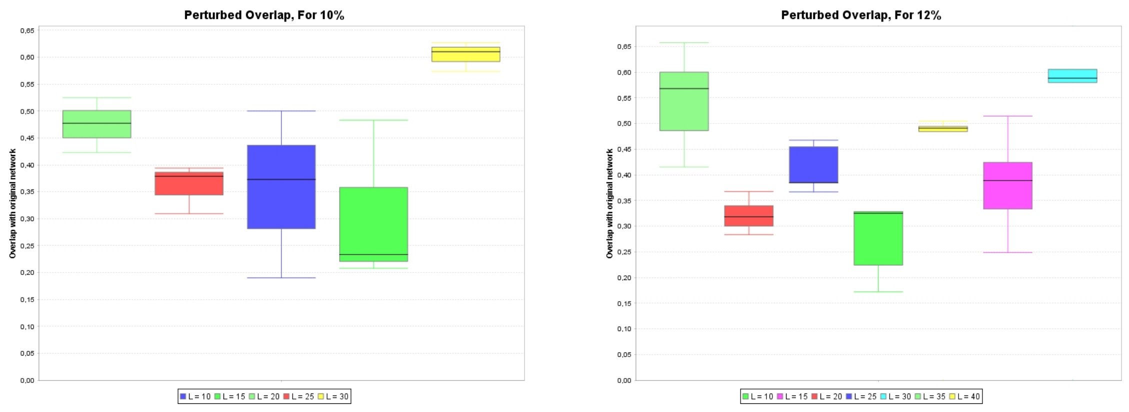
(b) L vs Nodes



(c) L vs Nodes, averaged

Figure 9: The three different standard charts.

In addition to these chart (Figure 9), if the run also contains perturbations, some boxplots are generated. These boxplots (Figure 10) shows the averaged overlap between the original run and the runs on the perturbed graph.



(a) Example with 10% perturbation.

(b) Example with 12% perturbation.

Figure 10: Example of the boxplots that are generated during a perturbation run.

6.1.4 Statistics

Since it is important for the users to get different statistics from the runs, a generalized interface called `IStatistics` (Listing 12) has been implemented. This interface exposes methods to obtain the statistical results, currently different types of overlap counts. At the time of writing, only one class, `BatchStatistics`, implements the interface. In addition to the interface methods, `BatchStatistics` also include a `calculate()`-method which performs the actual calculations needed to provide the results obtained from the interface.

Listing 12: **Interface - IStatistics**

```
public interface IStatistics {

    // unique identifier
    String getKpmID();

    // <chartname, chart>
    Map<String, IChart> getCharts();

    // <percent, overlap counts>
    Map<Integer, List<Double>> getPercentageOverlap();

    // <percent, <L, overlap>
    Map<Integer, Map<Integer, List<Double>>> getFixedPercentOverlap();

    // <percent, overlap-with-gold-standard-counts>
    Map<Integer, List<Double>> getGoldOverlap();

    // <percent, <L, overlap-with-gold-standard-counts>>
    Map<Integer, Map<Integer, List<Double>>> getFixedPercentGoldOverlap();
}
```

The first method of the interface (Listing 12), `getKpmId()`, is necessary as there could be multiple runs going on at the same time in the same instance of the application, e.g. the website. The value returned by this method is the unique identifier string for that exact run from which the results origin. During the runs, charts will be generated if it is a batch run, and `getCharts()` contains the map of those charts with their names as keys. The `getPercentageOverlap()`-method returns a percentage wise overlap map, with the percentage as key, and a list of overlap counts as value. `getFixedPercentageOverlap()` is a bit more complex, since it contains the results of counting overlap for each perturbation percentage as outer key, and for each varying L value as inner key. `getGoldOverlap()` returns the map with perturbation percentage as key, and the list of overlap counts for each run.

6.1.5 Task Monitors

One of the basic requirements for long running tasks in general, is that the user should be informed about the progress of the task, while it is running, preferably without blocking the user interface. To overcome this task, one has to ensure the task is not running on the GUI thread, and that the user interface can retrieve the information from the tasks. The first part is simple enough, as starting a new thread. The second part is a bit more complex, as different environments behave differently when it comes to retrieval of data from background threads. An approach to solve this for the web application will be explained in Section 6.4.1.

From the perspective of the long running task, the only knowledge it should have about the task monitors is the interface it exposes. For this project, the `IKPMTaskMonitor`-interface (Listing 13) was implemented.

Listing 13: **Interface - IKPMTaskMonitor**

```java
public interface IKPMTaskMonitor {
   public void setTitle(String title);
   public void setProgress(double progress);
   public void setStatusMessage(String statusMessage);
}
```

The runners (Section 6.1.6) takes an object implementing the interface as constructor input, and updates the different parts of the status. `setTitle()` updates the title of the task that is currently running. `setProgress()` takes a double as input, which should be between 0.0 and 1.0, that simulates the percentage of completion. `setStatusMessage()` sets the visible message to the consumer, here used to describe the current work being done.

In addition to the task monitor, there is also another interface listener for the runners. This is the `IKPMRunListener`-interface (Listing 14), which exposes two methods. `runCancelled()` is used when either of the two types of runners are canceled, for some reason, before it could produce a result set. The `runFinished()`-method is executed with the results of a run, after all charts and statistics has been generated. The `IKPMResultSet` contains the unique identifier for the run, the settings, results, charts and overlap results, all wrapped inside a generalized interface.

Listing 14: **Interface - IKPMRunListener**

```java
public interface IKPMRunListener {
   void runFinished(IKPMResultSet results);
   void runCancelled(String reason);
}
```

6.1.6 Runners

The `runners`-package contain the logic for performing the actual runs of the KPM algorithms. The classes `BatchRunner` and `BatchRunWithPerturbationRunner` both implements the Runnable interface, to get a simple threading integration. The `BatchRunner`-class (Listing 15) is the class that actually executes the algorithms, and interprets the results. It contains three main methods that were extended as a part of this project. As the class implements the Runnable interface, it must contain a `run()`-method which starts the process of running KPM, depending on the settings defined in the `KPMSettings`-class. `runBatch()` is called in `run()`, if the `KPMSettings` is setup as a batch run, with varying k and/or l values. Otherwise the `runSingle()`-method is called. Before saving the results for each run, the `BENRemover`-class is used. This class contains the implementation of the BEN removal algorithm described in Section 5.1.

Listing 15: **Class - BatchRunner**

```
public class BatchRunner implements Runnable{
   @Override
   public void run(){...}
   ...
   private void runSingle(){...}
   private void runBatch(){...}
}
```

A part of this project consisted of implementing the support for running multiple instances of KPM with different levels of perturbed graphs, without having to re-run the algorithms manually. To fulfill this requirement, the class `BatchRunWithPerturbationRunner` has been introduced. This class wraps and encloses instances of the `BatchRunner`, performing the perturbations with the help of the classes described in Section 6.1.1.

Listing 16: **Class - BatchRunWithPerturbationRunner**

```
public class BatchRunWithPerturbationRunner implements Runnable, IKPMTaskMonitor, IKPMRunListener{
   @Override
   public void run(){...}
   ...
   private void runWithPerturbation(){...}
}
```

The class implements both the `IKPMTaskMonitor` and the `IKPMRunListener` interfaces, as shown in Listing 16. This is because the class needs all results return by the `BatchRunner`-class.

6.2 KPM Cytoscape App

The first release of the KPM Cytoscape app was in 2012 [2], and has since been improved through several iterations to its current form in version 4.0 [4]. Over the course of this project, several changes to the package structure were made (Figure 11), to make it conform with moving the KPM core to its own library. An attempt to minimize the direct references to Cytoscape has been made, moving the usage and references to the `cytoscapehandlers`-package. This package contains the `CyActivator` (Listing 17), which is the base class for plugins/apps to Cytoscape. Since Cytoscape is panel-based, the implemented panels have their own package, call `gui.panels`, which is where all user interface is defined for the plugin. Next is the `interfaces`-packages, that contains all interfaces used across the plugin, for simplifying communication between classes. The connection to the kpm-core module is located within the `kpmhandlers`-package, in which handlers are defined for executing KPM and handling the results generated. Lastly, the `kpmhandler.mappers`-package, is the implementation of the mapper for the task monitor of Cytoscape and kpm-core.

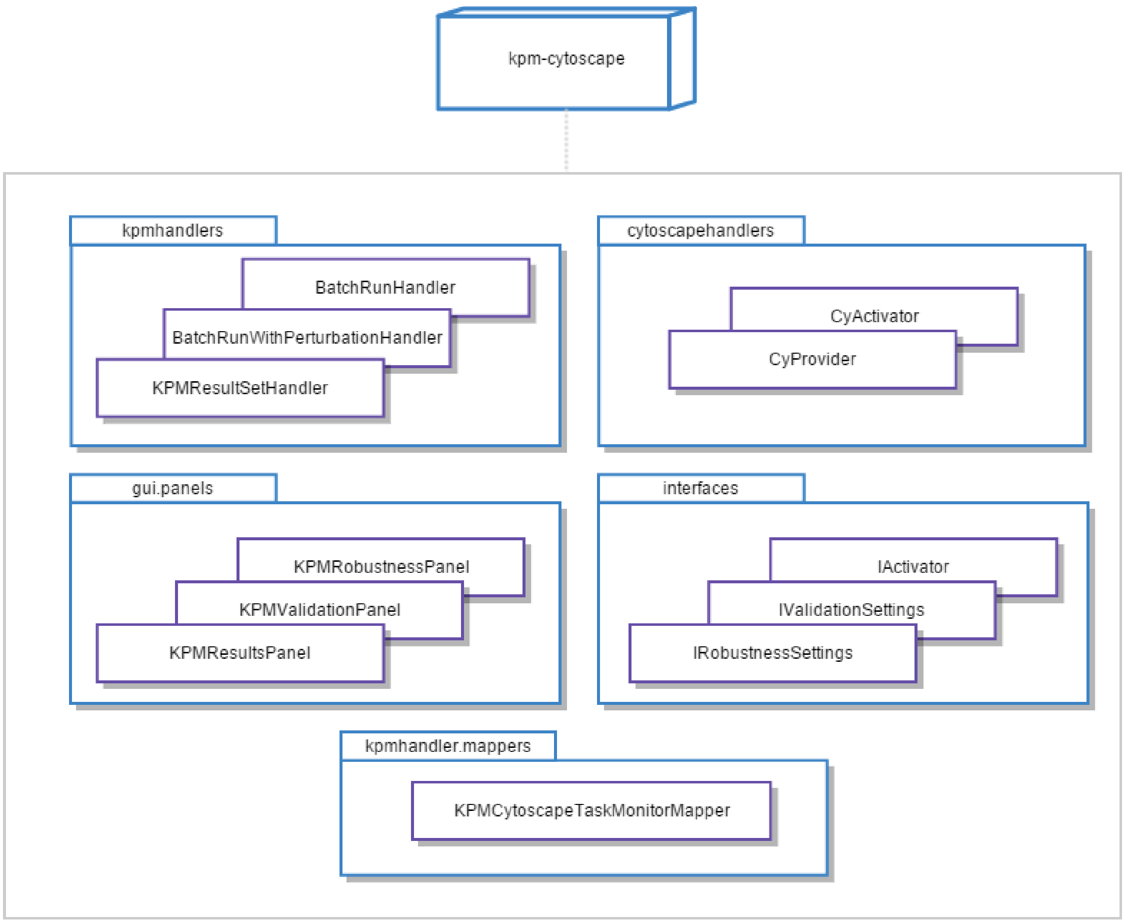


Figure 11: Diagram showing the general layout of the KPM Cytoscape plugin. It has been implemented trying to limit the usage of direct contact with the Cytoscape classes, to improve maintainability, by reducing the impact of breaking changes in Cytoscape.

To create a Cytoscape app, the application has to be wrapped in a JAR, with a class that extends `AbstractCyActivator`, by default called `CyActivator` (Listing 17). The `start()`-method is executed, when the plugin is started, and this is where the setup of the plugin begins. The different Cytoscape specific settings are defined here, the plugin services (panels) are registered, and the `CyProvider` is setup. `CyProvider` is a static class that contains pointers to all the Cytoscape specific objects, such as a network manager or a task manager. The `IActivator`-interface defines the two last methods described in Listing 17. It has the purpose of registering a JPanel, focusing it, or getting the instance.

Listing 17: **Class - CyActivator**

```
public class CyActivator extends AbstractCyActivator implements IActivator {
  @Override
  public void start(BundleContext bc){...}
```

```
    @Override
    public void registerAndFocusCytoPanel(CytoPanelComponent comp){...}
    @Override
    public CytoPanel getCytoPanel(CytoPanelName panelName){...}
  }
```

6.2.1 Cytoscape Task Monitor

Since Cytoscape comes with an implementation of a task monitor, it should not be known to the kpm-core module. Because of this, a wrapper class called `KPMCytoscapeTaskMonitorMapper` (Listing 18) has been implemented. The purpose of this class is to mediate the information given by KPM and relay it to Cytoscapes own task monitor. This means that kpm-core is completely separated from Cytoscape, and if Cytoscape changes its task monitor definition, this class is the only one that needs changing.

Listing 18: **Class - KPMCytoscapeTaskMonitorMapper**

```
  public class KPMCytoscapeTaskMonitorMapper implements IKPMTaskMonitor{
      private final TaskMonitor taskMonitor;

      public KPMCytoscapeTaskMonitorMapper(TaskMonitor taskMonitorParam){
          this.taskMonitor = taskMonitorParam;
      }

      @Override
      public void setTitle(String title) {
          this.taskMonitor.setTitle(title);
      }

      @Override
      public void setProgress(double progress) {
          this.taskMonitor.setProgress(progress);
      }

      @Override
      public void setStatusMessage(String statusMessage) {
          this.taskMonitor.setStatusMessage(statusMessage);
      }
   }
```

The way Cytoscape handles tasks is through the `DialogTaskManager`-class, which takes an object implementing the `AbstractTask`-interface as parameter. This interface defines a method `run()`, which has a `TaskMonitor`-instance as input. Examples of classes implementing the `AbstractTask`-interface will be described in Section 6.2.2.

6.2.2 KPM Handlers

To be able to control the flow between kpm-core and the Cytoscape plugin, we needed some appropriate wrapper classes to act as a mediator for the rest of the plugin. The two main mediators are the `BatchRunHandler` (Listing 19) and `BatchRunWithPerturbationHandler`, which both implements the `IKPMRunListener`-interface, and extends `AbstractTask`. The two `run()`-methods initiates instances of `BatchRunner` and `BatchRunWithPerturbationRunner`, respectively. A new instance of `KPMCytoscapeTaskMonitorMapper` is given here as input to the `runners`, such that we can see the progress of KPM in Cytoscape. Once the run has completed, the `runFinished()` method is called which starts a new thread with and instance of the `KPMResultsSetHandler`. The `KPMResultsSetHandler` mediates the results gotten from the run, creates a charts panel and initiates the construction of the `KPMResultsTab`.

Listing 19: **Class - BatchRunHandler**

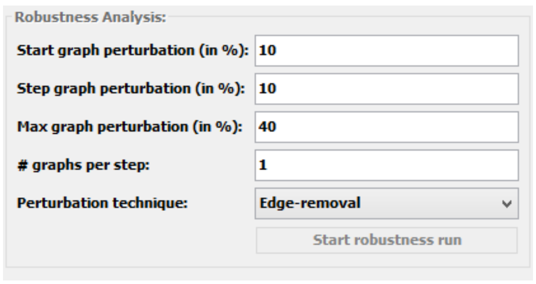
```
public class BatchRunHandler extends AbstractTask implements IKPMRunListener{
    @Override
    public void run(TaskMonitor tm) throws Exception {
        KPMCytoscapeTaskMonitorMapper mappedListener = new KPMCytoscapeTaskMonitorMapper(tm);
        ...
        this.batcher = new BatchRunner(mappedListener, ...);
        this.batcher.run();
    }

    @Override
    public void runFinished(IKPMResultSet results) {
        ExecutorService executor = Executors.newCachedThreadPool();
        KPMResultSetHandler handler = new KPMResultSetHandler(results);
        executor.submit(handler);
    }
}
```

6.2.3 Robustness- and validation analysis

To enable robustness and validation analysis. Two components (Figure 12) were added to the plugin, in the Run-panel. The first one is called `KPMRobustnessPanel` (Figure 12a), which defines the perturbation parameters used in the robustness run. These settings comes in addition to the existing ones, and uses them as well. This way the settings only have to be set in a single "page". As seen on the figure, there are settings for both start, step and max perturbation, as well as the number of graphs per step, and the perturbation technique to be used.

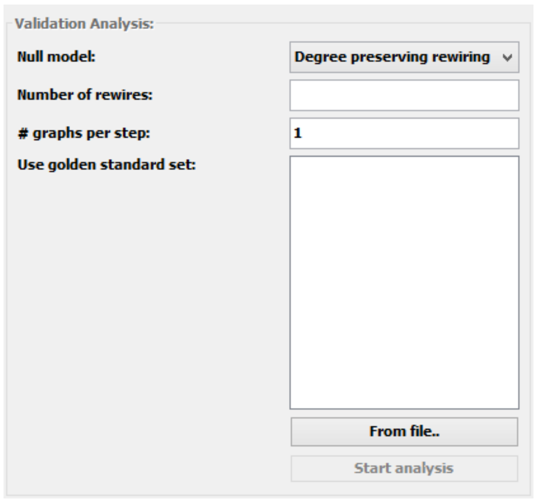
The other component is called `KPMValidationPanel` (Figure 12b), which defines a null model (perturbation), and if the "Degree preserving rewiring" is chosen, the number of rewirings is also visible to set. Then there is the number of graphs per step, and lastly, an input for a list of gold standard nodes for which overlap should be counted.

Once the setup has completed for either of them, and other settings have been appropriately set, the start-buttons become clickable, which will start the run, once pressed. The rest of the process is automated.


Robustness Analysis:

Start graph perturbation (in %): 10
Step graph perturbation (in %): 10
Max graph perturbation (in %): 40
# graphs per step: 1
Perturbation technique: Edge-removal

(a)


Validation Analysis:

Null model: Degree preserving rewiring
Number of rewires:
# graphs per step: 1
Use golden standard set:

(b)

Figure 12: This figure shows the added feature for robustness and validation analysis in the KPM Cytoscape plugin.

6.3 KPM Standalone

The standalone version of KPM originally contained logic for performing the KPM calculations by itself. This was changed with the implementation of `kpm-core`. Now `kpm-standalone` (Figure 13) references the `kpm-core` project, exactly like the Cytoscape plugin. This console application can be configured in three different ways: a settings file, command line arguments, or a combination of the two. This requires the classes found in the `parsers`-package, as it is necessary to parse the setup.

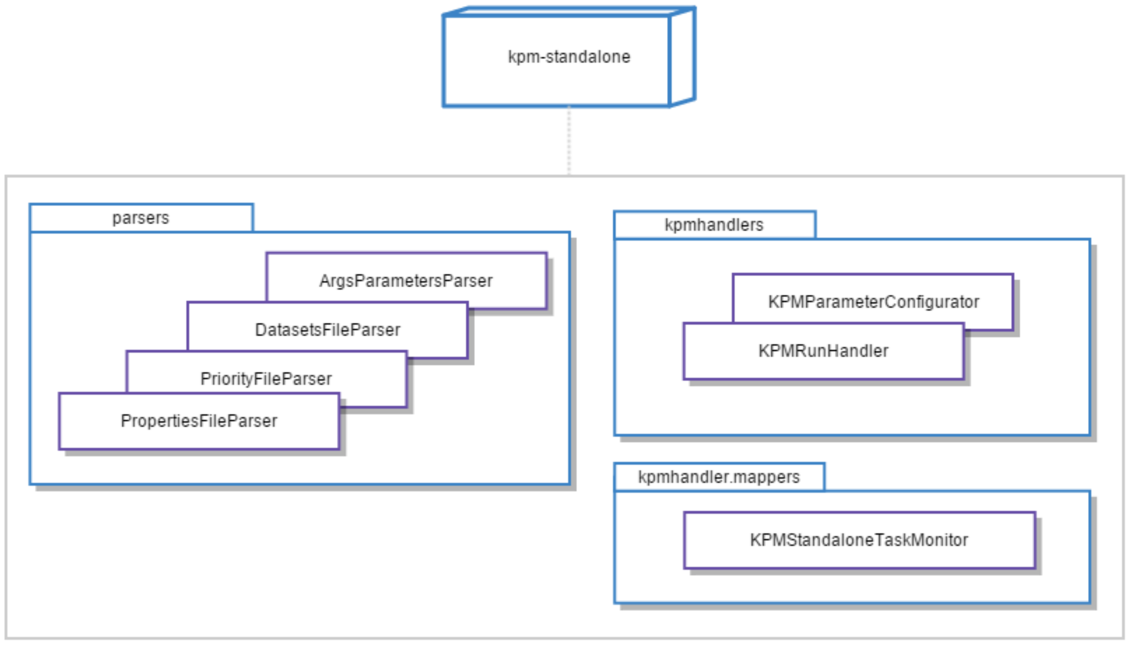


Figure 13: A visualization of how the `kpm-standalone` module is build up. Here is only shown the parts implemented as a part of this dissertation, so it contains more than what is shown.

Much like the Cytoscape plugin, the connection to `kpm-core` is mediated through a `kpmhandlers`-package. It also contains a `mappers`-package in which a simple task monitor is implemented. The largest package is the `parsers`-package. This contains all the parsers for the different types of inputs that this application can take. It should be noted that the main work for the `parsers`-package was performed by Nicolas Alcaraz, over the course of his own project work with KPM. As a part of this dissertation, the parsers were merely adjusted for the implemented changes to KPM.

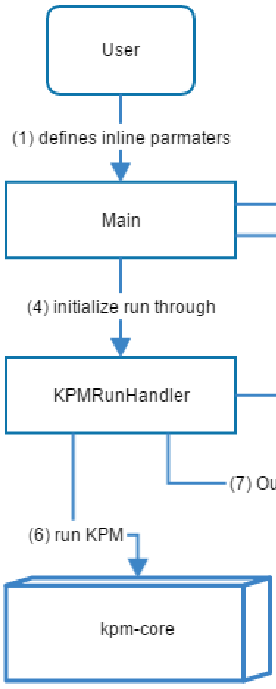

User
(1) defines inline parmaters
Main
(2) get parsed parameters
ArgsParametersParser
(3) get input files from parameters
InputFileParser
(4) initialize run through
KPMRunHandler
(5) configure KPM parameters
KPMParameterConfigurator
(7) Output results to User
(6) run KPM
User
kpm-core

Figure 14: A workflow of what happens when a user initiates the standalone version, with command line arguments.

On Figure 14 it is shown what happens in `kpm-standalone`, when executed with command line arguments. It can be described as follows:

(1) In `kpm-standalone`, input parameters are given via command line arguments to the `main`-class.

(2) The `main`-class utilizes two parsers. The `ArgsParametersParser` converts the inline arguments to a `Parameter` object.

(3) The resulting parameters are then used as input for the `InputFileParser`, which parses all the input files, e.g. graphs and datasets.

(4) The `main`-class feeds these parameters to a `KPMRunHandler`, which handles the run.

(5) The `KPMRunHandler` uses the `KPMParameterConfigurator` to configure the KPM parameters, which is an object type of its own, from the input parameters. Explanation of the run handler is explained in Section 6.3.2

(6) The configured KPM parameters are then fed to `kpm-core`, and the run is executed.

(7) The results from the run are then saved on the disk to somewhere the user defined.

6.3.1 Parsers

The parsers serve the purpose of parsing the input from the user, into settings for KPM, therefore several parsers have been implemented.

`ArgsParametersParser`
This parser (Figure 20) is used to parse the command line arguments. It contains, like

all the other parsers, a constructor with a reference to the current KPM Settings, which will be manipulated. It also contains a `parse()`-method, which takes the command line arguments as parameters, in addition to an instance of some parameters. In addition to handling all available line arguments, it is also able to display the contents of the `readme` file, as it is defined in the `printHelp()`-method.

`DatasetsFileParser`
The purpose of the `DatasetsFileParser` is to convert a dataset file into a case exceptions map, and a matrix file map. The matrix file map is to identify the input file later on. Currently this class is used by the `ArgsParametersParser` to parse the input dataset files defined by the command line arguments.

`InputFileParser`
As the name implies, this parser handles the conversion of all the input files for the run. The main purpose of this parser is to convert the files containing positive or negative nodes into their corresponding KPM setting.

`PriorityFileParser`
The `PriorityFileParser` is more a utility, as it parses a generic text file into a list of strings, with a string for each line. This parsers is used to parse the files with positive and negative nodes by the `InputFileParser`.

Listing 20: **Class - ArgsParametersParser**

```java
public class ArgsParametersParser {

  private volatile KPMSettings kpmSettings;

  public ArgsParametersParser(KPMSettings settings) {
    this.kpmSettings = settings;
  }

  public Parameters parse(String[] args, Parameters params) throws Exception { ... }

  private static void printHelp() { ... }
}
```

6.3.2 Run handler

Once the parsers have finished their jobs, it is the job of the run handler, named `KPMRunHandler` (Listing 21), to mediate with `kpm-core` and start the run. The `public` method `runBatch()` collects all the parameters from the parsers and based on the settings, executes either the `runStandard()`- or the `runBatchWithPerturbation()`-method.

In the `runStandard()`-method, the runs without perturbations are initiated, which mean an instance of `BatchRunner` is created and started. In the `runBatchWithPerturbation()`-method, an instance of `BatchRunWithPerturbationRunner` is created and started. Once the runs has completed, be it with or without perturbaton, the `runFinished()`-method is executed, which saves all the standard charts (Figure 9) to the results folder.

Listing 21: **Class - KPMRunHandler**

```java
public class KPMRunHandler implements IKPMRunListener{
    public void runBatch(Parameters params){ ... }
```

```
        private void runBatchWithPertubation(Parameters params){ ... }

        private void runStandard(Parameters params){ ... }

      ...

        @Override
        public void runCancelled(String reason) { ... }

        @Override
        public void runFinished(IKPMResultSet results) { ... }
    }
```

6.4 KPM Web application

The implementation of a web application/service is one of the major contributions of this thesis. The platform chosen is the Groovy/Grails MVC(S) framework (Figure 15). This platform was chosen due to its direct support for external Java JARs, and because it is a state-of-the art framework for both rapid and robust web application development. The implementation of MVC also includes a Service-type object. This object type is a stateless, transactional object in which business logic is supposed to be placed. This is also where the mediation to `kpm-core` takes place.

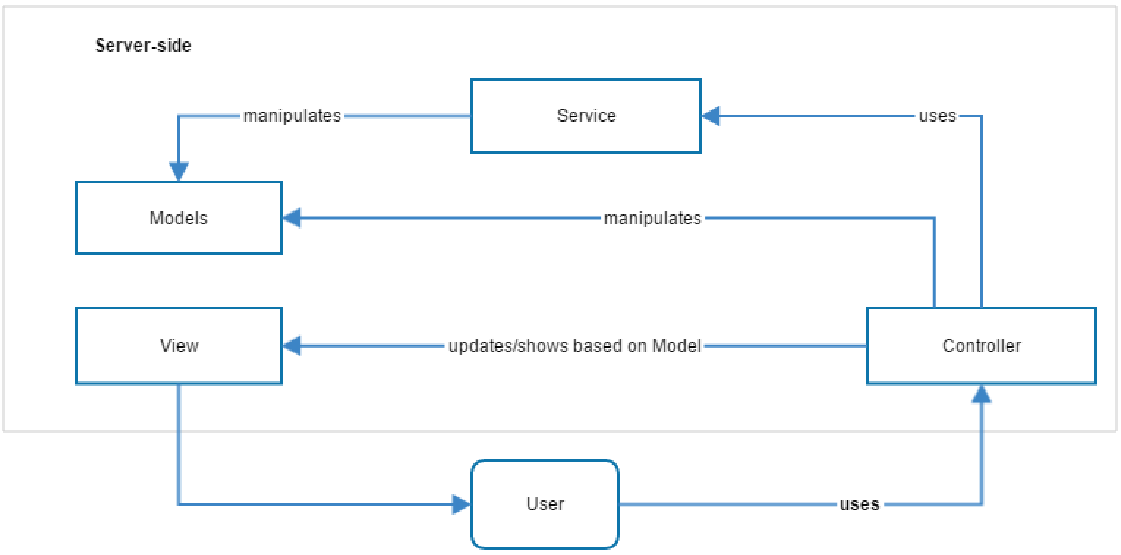


Figure 15: Groovy/Grails MVC(S) workflow. It is shown how the MVC framework works by having the user exposed to options through `views`, which execute methods on the `controllers`, that in turn manipulates data-`models`. An important difference to other MVC-frameworks is the addition of the `service`-type object.

The `kpm-web` project is divided into several packages (Figure 16), each containing important aspects of the project as a whole. The `RunKPM`-class manages the progress in which the user is shown the `runKPM`-views, that sets up the settings class, `RunParameters`, for running KPM in the web application. The `QuestsController` manages the content shown in the `questsAttachedToID`-view, which is a page showing all quests (progress of run) related to an input parameter (run ID). The `KpmService`-class is the mediator to `kpm-core`, in which `KpmWebTaskMonitor` is used as the task monitor, updating the corresponding `Quest`. The

"Quests" (threading) concept will be explained in Section 6.4.1, which also goes into the details of the functionality for the `QuestService`. The results of a KPM run is picked up by the `KpmListener`-service, saving graphs and charts to the database which can later be shown to the user.

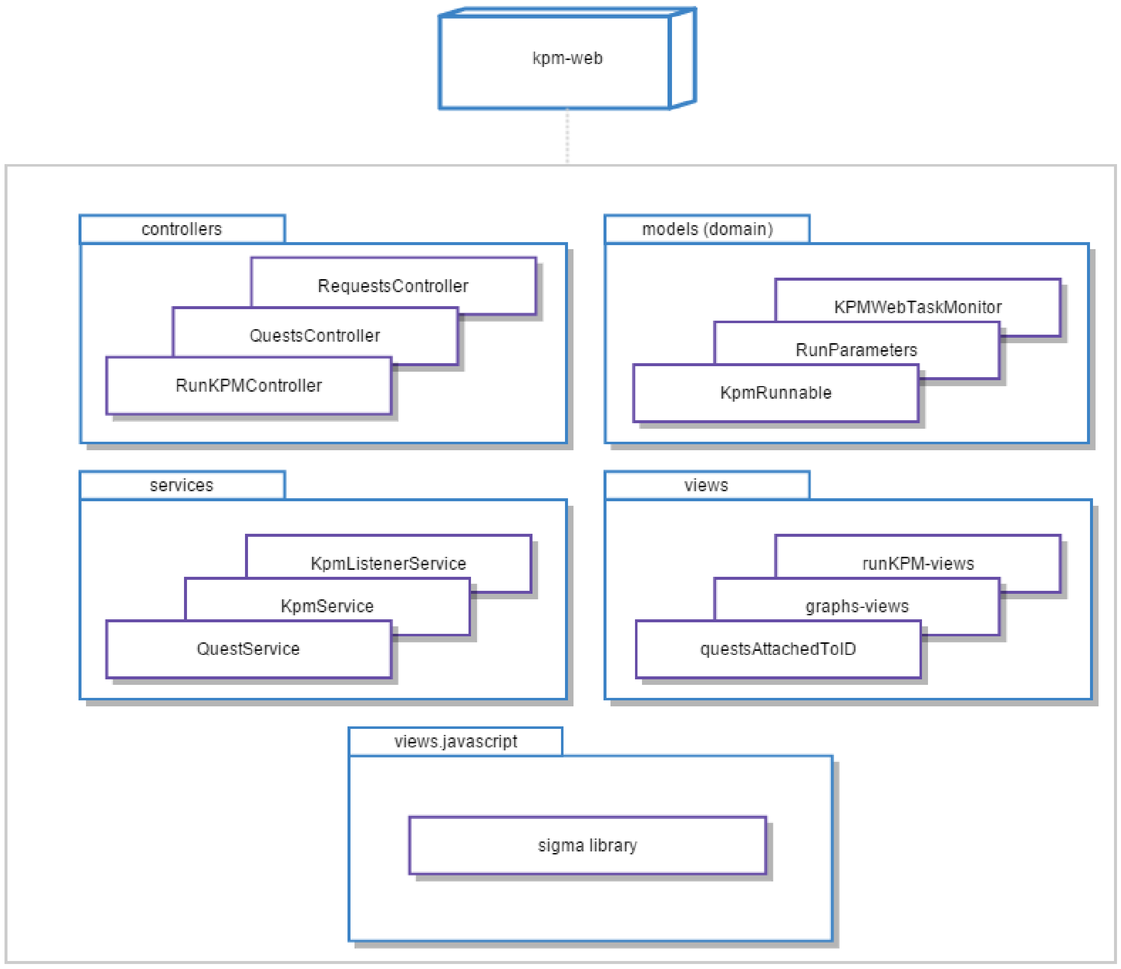


Figure 16: KPM Web module UML packages. This figure shows how the `kpm-web`-module is constructed. As it is an MVC(S) project, it is divided into Models, Views, Controllers, and Services. The content of the `views.javascript`-package is the JavaScript network graph library and its dependencies.

6.4.1 Threading and tasks (quests)

Threading in web applications is different from both console, and Swing applications. Because a browser is used as a mediator between the server and the user, it is necessary to be able to identify every run and thread, if communication is needed. To this end no Java, Groovy, or Grails plugins were found, that could fulfill the need for running tasks server-side, with something that came close to a "task monitor", in the sense it has been used in this dissertation.

Therefore, a new concept termed "Quests" (Figure 17) was introduced. Which is a specification of a task progress, with another name to avoid confusion with other libraries. A `Quest` can hold information like the run ID, a status message, a title, and ID of the `RunParameter` for the given run. A `Quest` is saved to a database, and updated from a background thread running KPM.

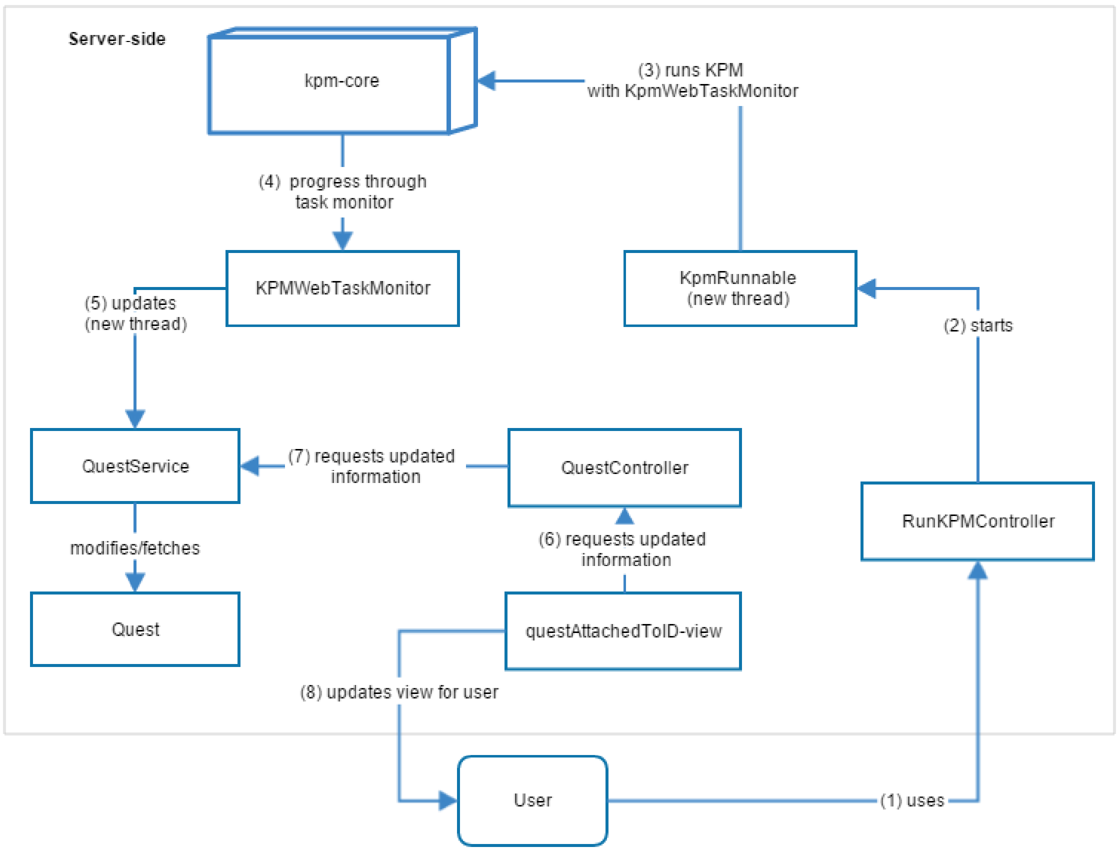


Figure 17: KPM Web Quests (tasks and threading). Flow diagram showing how quests is used to mediate updates between the background tasks running KPM, and the browser.

The workflow for Quests are used and updated to show the progress of a KPM run, is as follows (Figure 17):

(1) The user initiates a new KPM run by setting up the parameters, through the views for the `RunKPMController`.

(2) Once the setup has completed, `RunKPMController` starts a new instance of `KpmRunnable` on a new thread.

(3) `KpmRunnable` starts the KPM run, using an instance of `KpmWebTaskMonitor` as the task monitor parameter.

(4) Since `KpmWebTaskMonitor` is used as the task monitor, KPM will use it to notify about the progress of the run.

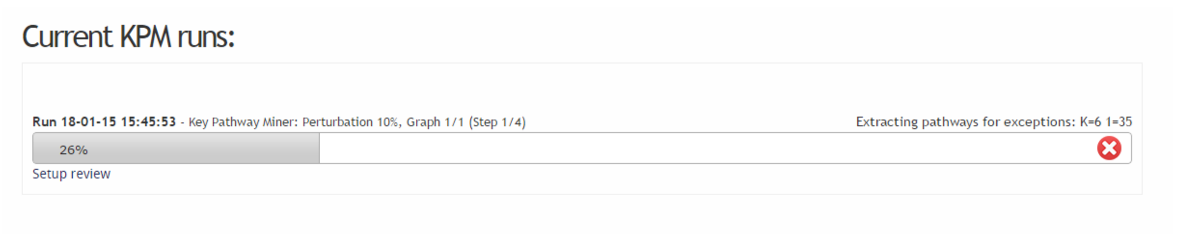
(5) Because of the way locking occurs in the database, saving often forces the updated `Quest` to not be flushed, for other threads to see, until the updating thread has terminated. To overcome this problem, a new thread is initialized with the only purpose to update the `Quest`. This way, updating is flushed every time an update has completed.

(6) The user can see the progress of a run through the `questAttachedToID`-view, which shows all quests attached to a user ID. The `questAttachedToID`-view uses asynchronous calls to the `QuestController`, requesting updated information about the quests.

(7) The `QuestService` pulls the updated, and flushed, information from the database, and returns it to the `QuestController`.

(8) The `questAttachedToID`-view updates the progress of the returned views, providing a near real-time progress update to the user.

The implementation effectively provides the user with feedback from KPM running on a thread on the server, in a manner that sufficiently resembles a progress bar with status text and progress indicator. This way, the look and feel of KPM is similar in both the Cytoscape plugin and the web application. This ability to see how the run progresses without having to reload the page is important for the user friendliness. This is achieved through asynchronous calls in JavaScript. On Figure 18 is shown a snapshot of a run progress (quest) status.


Current KPM runs:

Run 18-01-15 15:45:53 - Key Pathway Miner: Perturbation 10%, Graph 1/1 (Step 1/4)

Extracting pathways for exceptions: K=6 1=35

26%

Setup review

Figure 18: Status bar showing how a run, executing in the background on the server, is progressing.

6.4.2 Network graphs

The Sigma library was used to show the resulting network graphs, as explained in Section 4.4. This library provided the necessary functionality to customize the network graphs to suit the need for this thesis. The graphs can be fetched by dragging the sliders for k, l and computed subgraphs, as shown on Figure 19. In addition to providing the user with a graphical overview of the solution, each node is linked to the NCBI [20] website. If a node is clicked, then a new window opens to the corresponding page on the NCBI website, so the user can see all gathered knowledge about that specific node.

In addition to a normal network graph, a union graph (Figure 19) can be selected, which is the union of all result nodes for all solutions KPM reported for a given k and l value. In the normal graph, all nodes are gray, but in the union graph they are colored to mark the number of occurrences across the different solutions. The color scale reaches from green (low number) to red (high number). If the number of occurrences is greater than one, it will be shown in parenthesis after the node id.

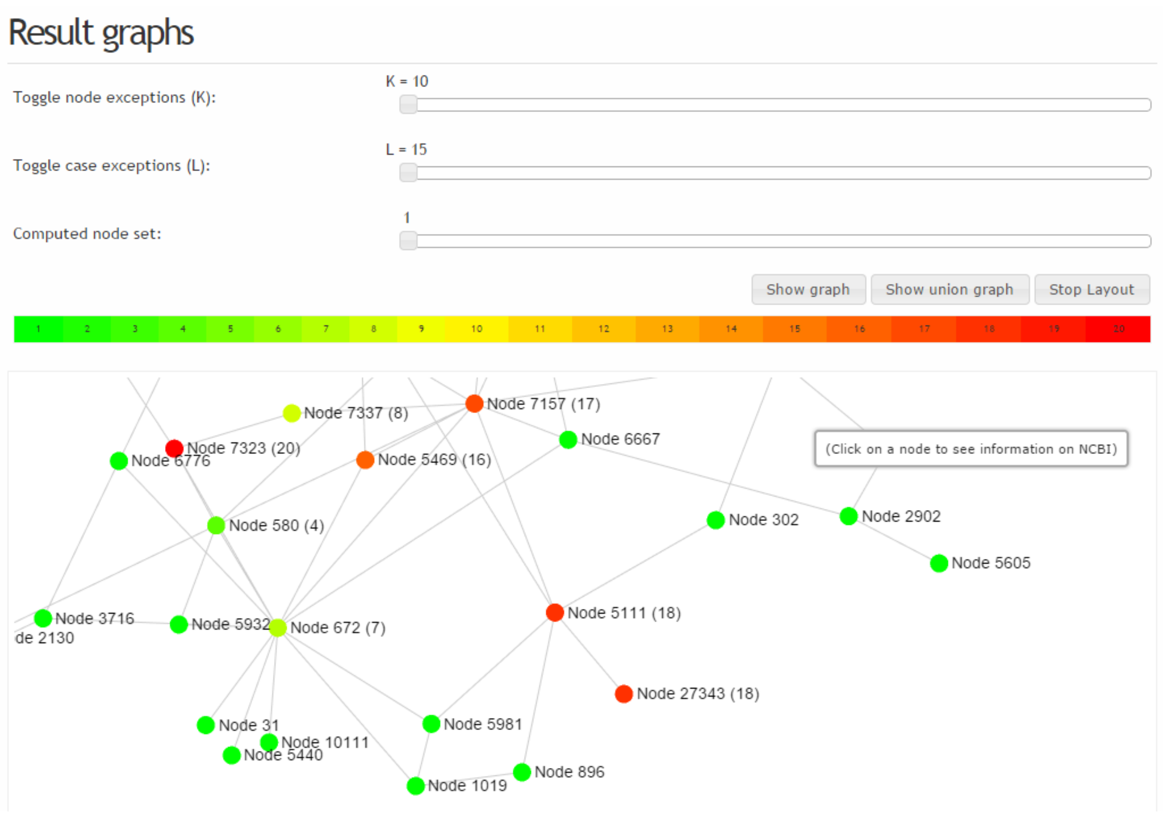


Figure 19: Snapshot from the KPM web application, of the KPM solutions union graph.

6.4.3 KPM as a web service

To fulfill the requirement for supporting integration, it was necessary to enable setup and connection to KPM from other platforms than a web browser. This was done by first utilizing the structure of the MVC framework. By default it supports requests to a web page, and the Controller behind it, can do any task it is programmed to. In the `kpm-web` project, the `kpmJSON()`-method of the `RequestController` is used to facilitate a connection between the web server and any language supporting JSON, and which is able to perform web requests (Figure 20).

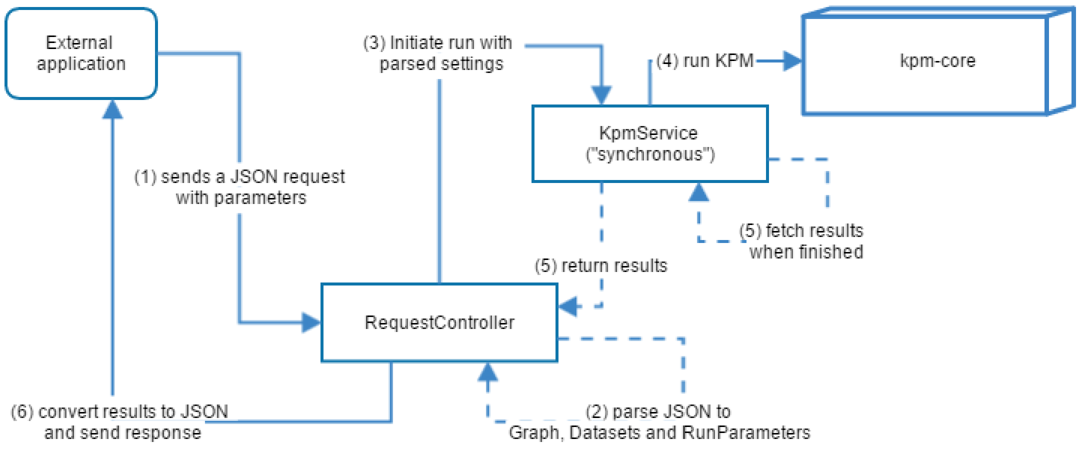

External application
(1) sends a JSON request with parameters
(3) Initiate run with parsed settings
(4) run KPM
kpm-core
KpmService ("synchronous")
(5) fetch results when finished
(5) return results
RequestController
(6) convert results to JSON and send response
(2) parse JSON to Graph, Datasets and RunParameters

Figure 20: KPM Web Service flow. This figure shows how a request from an external application is handled by the service, and how the results are obtained, converted to JSON and returned to the sender.

Synchronous execution

An external application can connect and use the web service in the following manner for synchronous execution (Figure 20):

(1) The external application sends a JSON formatted request to the `RequestController`, which is exposed by an URL like `http://kpm-web.com/request/kpmJSON`.

(2) The `RequestController` parses the JSON to graphs, datasets and other parameters, only continuing if all settings, needed for a run, is available.

(3) The `RequestController` executes a `startSynchronous()`-method on the `KpmService` with the settings that were parsed.

(4) The `startSynchronous()`-method on `KpmService` starts a KPM run, which simulates a synchronicity.

(5) The `startSynchronous()`-method actually starts a normal run, which is executed on another thread, and then it starts looping with a one second interval. Within the loop, the progress of the run is checked with the help of the `Quest` attached to the run. Once the run has completed, the results are returned to the `RequestController`.

(6) The `RequestController` converts the resulting graphs into JSON elements and sends a response to the listening external application.

With this implementation, KPM will supporting integration into any external application which has internet access and supports the JSON format.

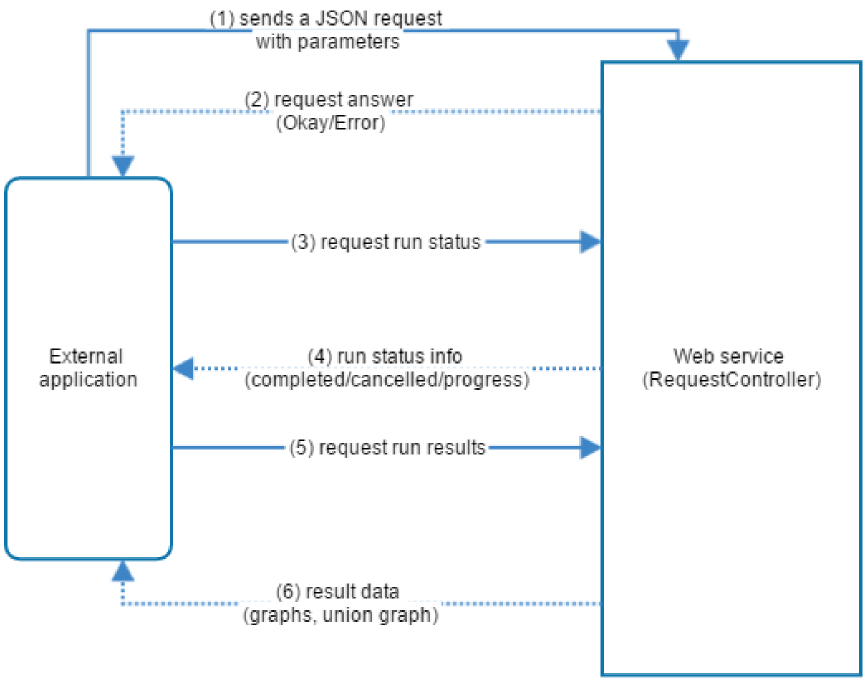

(1) sends a JSON request with parameters
(2) request answer (Okay/Error)
(3) request run status
External application
(4) run status info (completed/cancelled/progress)
Web service (RequestController)
(5) request run results
(6) result data (graphs, union graph)

Figure 21: KPM Web Service asynchronous flow. This figure shows how an external application requests an asynchronous KPM run, and fetches status updates and results.

Asynchronous execution

Asynchronous execution of KPM is also possible via the web service. An external application can perform asynchronous execution in the following manner (Figure 21):

(1) The external application sends a JSON formatted request to the `RequestController`, which is exposed by an URL like `http://kpm-web.com/request/kpmJSONAsync`.

(2) The `RequestController` parses the JSON input, and starts the run on a new thread, via `start()`-method on `KpmService`, allowing a response to be send back to the external application. The response contains information like the run ID, quest ID, and a status message telling if the run was started.

(3) While KPM is running, the external application can request run status updates via the URL `http://kpm-web.com/request/kpmRunStatus?questID=<id of a quest>`.

(4) This will make the `RequestController` pull the `Quest`-object with the parameter ID from the database, and return a JSON formatted response indicating percentage progress, and boolean values indicating whether the run has completed, is canceled or even exists. During a successful run, while it has not completed, step (3) and (4). Once the status returns completed, the next step will be taken.

(5) Once the run has finished, e.g. (4) returns that the run has completed, a request can be send to `http://kpm-web.com/request/kpmResults?questID=<id of a quest>`.

(6) This will make the `RequestController` return all result graphs, including the union graph, in a JSON format.

6.5 R as client for KPM web service

The R statistical framework (Section 4.3) was chosen as an example client for the KPM web service (Section 6.4.3), as it supports both web requests and encoding/decoding JSON elements, with the help of plugins like RCurl and RJSON.

To use the web service, it was necessary to create some wrapper methods which could handle converting the inputs, sending the request and returning the response. The first method needed is one for converting the content of a file into a base64 encoded string (Listing 22).

Listing 22: **R code - base64EncodeFile(fileName)**

```
base64EncFile <- function(fileName){
    return(base64(readChar(fileName, file.info(fileName)$size)))
}
```

The `base64EncodeFile()`-method is used to convert both graphs and datasets to a base 64 format. Next is used the `toJSON()` from the RJSON-package. This method takes any list, and object, and serializes them to the JSON format (Listing 23). Once the method has completed, the `graph` variable will now contain a JSON serialized object with values for "name", "attachedToID" and "contentBase64".

Listing 23: **R code - Graph to JSON**

```
graph1 <- base64EncFile(graphFile1)
graph <- toJSON(c(name=graphFile1, attachedToID=ATTACHED_TO_ID, contentBase64=graph1))
```

The last type of object needed to use the KPM service is to parse the KPM settings into a JSON format (Listing 24). Here it is possible to define almost all the same parameters as the other KPM programs. Server-side there is a check on correctness of the parameters, which returns a proper failure response to the client. This way, there is always ensured correct settings.

Listing 24: **R code - KPM settings to JSON**

```
KPMsettings <- toJSON(
list(
    parameters=c(
      algorithm="ACO",
      strategy="INES",
      ...
        )),
    perturbation=list(c(
          technique="Node-removal"
          ...
        )),
    withPerturbation="true",
    ...
    ))
```

Once the settings have been serialized to JSON format, the post to the web service can be performed (Listing 25). The whole process of connecting with the service and parsing the results is done in two lines. First line sends the data to the web service via RCurl's `postForm()`-method. The second line uses RJSON's `fromJSON()`-method, which converts JSON-formatted data into R type objects.

Listing 25: **R code - Call to KPM Service**

```
result <- postForm(url, kpmSettings=kpmSetup, datasets=datasetList, graph=inputGraph)
jsonResult <- fromJSON(result)
```

The results from the KPM run is the list of result graphs, obtained from the run, along with information about the K and L values.

6.5.1 Obtaining standard graphs from service

One of the paramaters of the KPM settings is the name of the graph. if the user wants to use one of the standard graphs provided, it is necessary to obtain the list of their names. This is done via the `RequestController`'s `graphsAsJSON()`-method, which returns just that.

7 Results

In this section, the extensions to KPM created as a part of this thesis will be evaluated. The four different interfaces of KPM will be tested for functionality, and compared against each other. Due to the way KPM works, some variations are expected, however the results should be closely related.

The common settings for both analysis tests, across all applications, are shown on Table 3. The INES strategy with the Greedy algorithm was chosen. The dataset "coad-exp-up" mentioned in Section 4.6.6 will be used with the "graph-ulitsky-entrez" mentioned in Section 4.6.5. The number of computed pathways has been set to 20, and unmapped nodes will be added to a positive list.

Parameter	**Value**
Dataset	COAD-EXP-UP-p.0.05.dat
Algorithm	Greedy
Strategy	INES
Graph	graph-ulitsky-entrez.sif
Treatment of unmapped nodes	Add to positive list
# computed pathways	20
Validation set	COAD-VAL-ENTREZ.txt

Table 3: Common test parameters

7.1 Robustness analysis tests

The purpose of a robustness test is to investigate how much the results obtained through perturbed networks change, in comparison to unperturbed networks. This is to see how much information KPM can recover when using incomplete or noisy data. That way it is possible to evaluate how much perturbation KPM can tolerate before the quality of the results starts to decrease to an unacceptable level.

The robustness analysis setup is shown in Table 4. The output charts of each run are depicted in Appendix B.2, where they are aligned according to type, so they can be compared. Please note, that since kpm web service is a part of the kpm web application, only the results of running the web application will be shown.

Parameter	**Value**
Node exceptions (K)	(range) from 8 to 12, with step 2
Case exceptions (L)	(range) from 10 to 30, with step 5
Perturbation	Node remove, from 2% to 12%, with step 2%

Table 4: KPM robustness test parameters

The tests were performed through the web application. As expected it shows how the overlapping amount decreases as the perturbation percentage increases. The tolerance level differs for each individual research case and therefore a universal threshold cannot be found. As an example, consider 0.5 as the Jaccard coefficient (Eqn. 2) lower bound. The 2% perturbation on Figures 22 clearly shows an average at above 0.6. At 6% perturbation the lower bound has been reached, and it can be concluded that for this network and dataset, with the settings provided, KPM accepts between 5% and 6% perturbation of the graph. The remaining charts can be found in Appendix B.2.

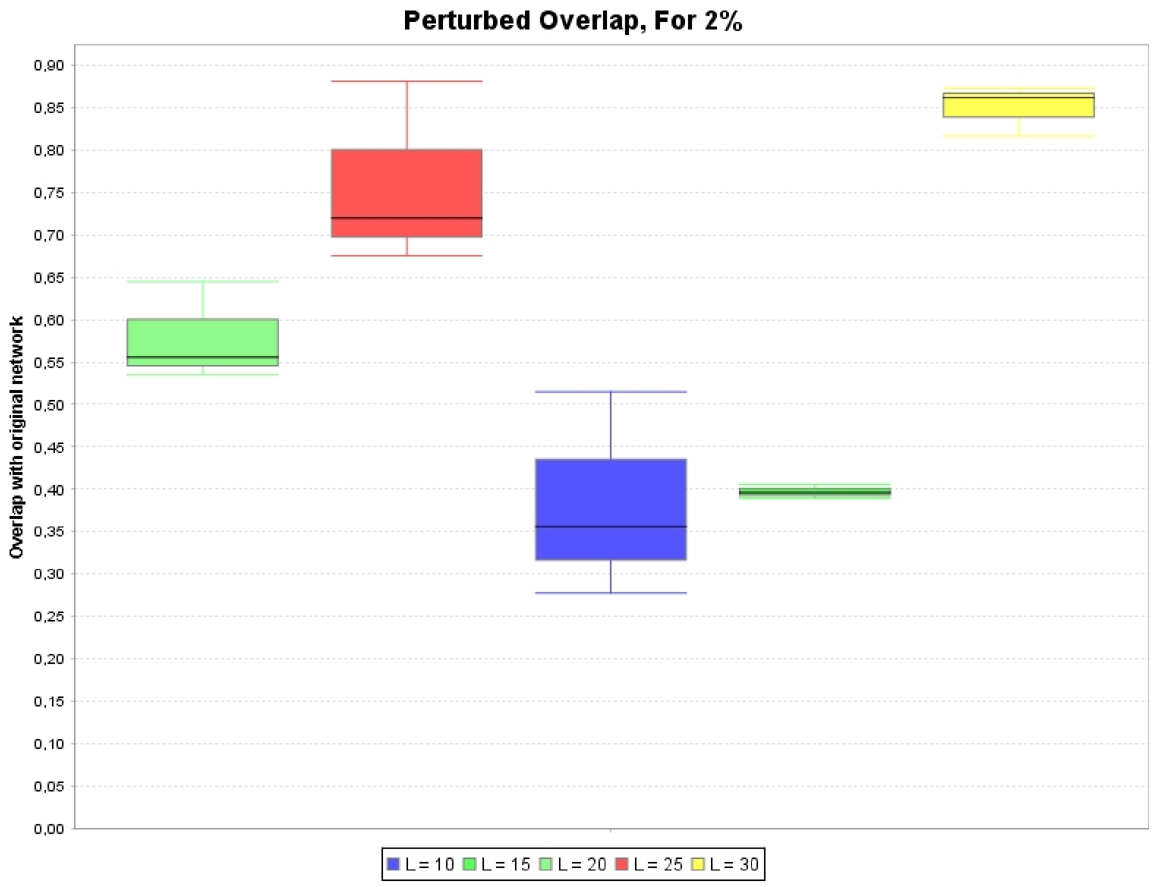


Figure 22: Output of robustness analysis, 2% perturbation

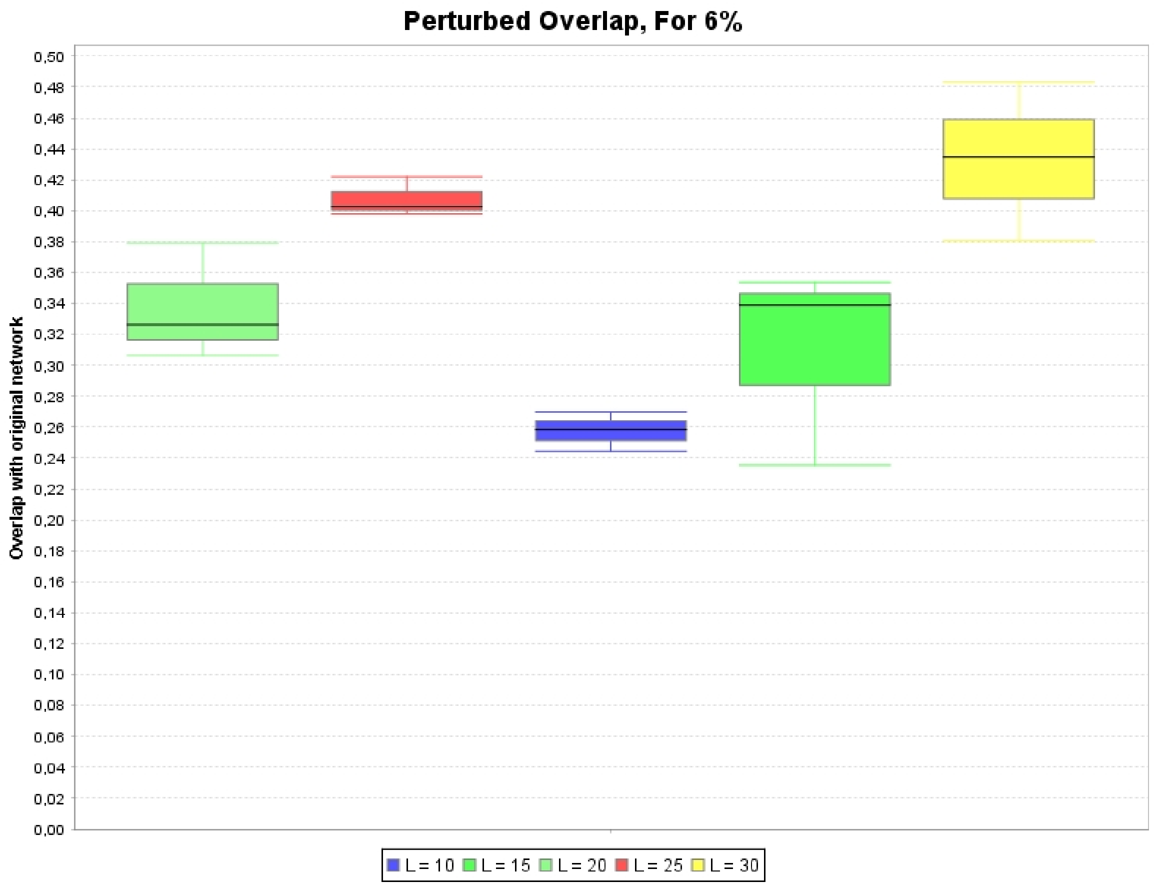


Figure 23: Output of robustness analysis, 6% perturbation

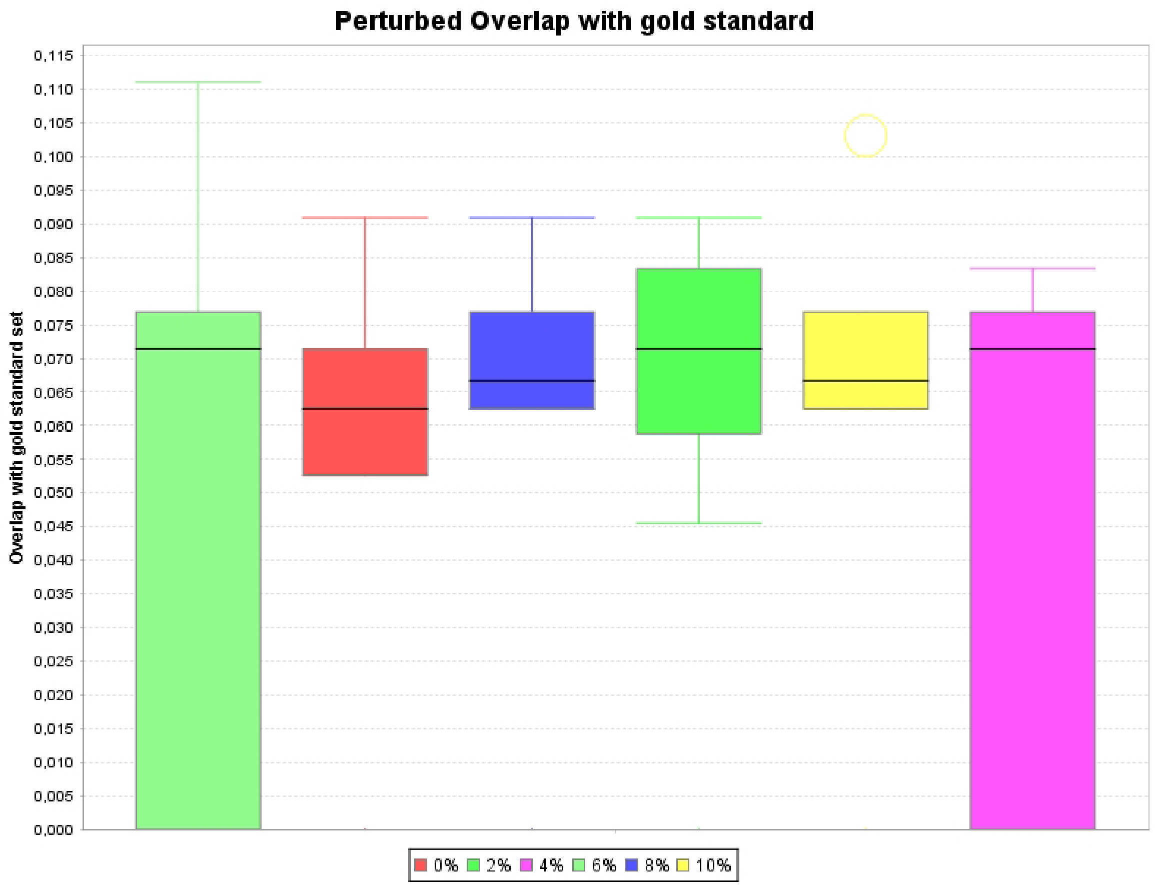
7.2 Validation analysis tests

The purpose of a validation analysis test is to see how many nodes from the results obtained through KPM overlap with a gold standard or pre-defined node list. In addition, degree preserving edge rewiring was implemented to assess the robustness of the validation analysis. The validation settings are specified in Table 5.

Parameter	**Value**
Node exceptions (K)	(range) from 8 to 12, with step 2
Case exceptions (L)	(range) from 10 to 20, with step 5
Perturbation	Edge rewire, from 2% to 10%, with step 2%
Gold standard set	COAD-VAL-ENTREZ.txt

Table 5: KPM validation test parameters

The test was executed on the web application, and the result showing the overlap with the gold standard is shown in Figure 24. The figure illustrates the overlap between the results and the gold standard as a boxplot, calculated with the Jaccard coefficient (Eq. 2). Because the chosen gold standard set is fairly large in comparison to the result set size, a lower coefficient was expected. It is clear from the figure, that even though the coefficient is low, it averages greater than zero for all perturbed runs. Therefore, it can be concluded KPM successfully recovers gold standard nodes, even with a 10% perturbation of the input graph.



(a)

Figure 24: Charts showing how well the percentages overlap during. The full size chart can be found in Appendix B.3.

7.3 Run-time test

In addition to demonstrating the new robustness and validation analysis, it is also important to show how these addition impact the overall run-time of KPM. Since there was no changes to the algorithms or strategies themselves, the run-time for a single KPM run is presumed to be the same as before, bounded by $O(|V|^k + |V|^3 + |E|)$ [2], as stated in Section 4.6.4. Since all implementations share the same library, the same running time can be assumed.

The setup for the test is shown in Table 6. Like other tests performed here, it shares the parameters of Table 3. The output of the run-time test is shown in Figure 7 and indicates that the nearly all of run-time has to be attributed to the KPM algorithm.

Parameter	**Value**
Node exceptions (K)	20
Case exceptions (L)	(range) from 20 to 30, with step 10
Perturbation	Node remove, 10%

Table 6: KPM run-time test parameters, for standalone version

	Execution time	
KPM algorithm (batch)	43996.9	ms (averaged)
Perturbation (per perturbation)	2292.3	ms (averaged)
Generating statistics and charts (per run)	< 1	ms
BEN removal	< 1	ms

Table 7: The actual run-times for the highlighted parts of KPM.

7.4 Web service integration test

In systems biology, there is a need for enrichment tools, to analyze gathered data. While there are tools which provide a web-interface, there are none with an implemented web service. The web service will make it possible for non-Java programmers to write a very short program, or script, like the R client described in Section 6.5 to access the functionality of KPM.

With the implementation of a website, KPM is now easily available to biologists, so they can perform the necessary enrichment analyses to their own projects, without the need to install Cytoscape. Moving away from client-side processing also means that a small workstation, or even a netbook or tablet, could be used where a powerful PC was favored before to obtain results fast.

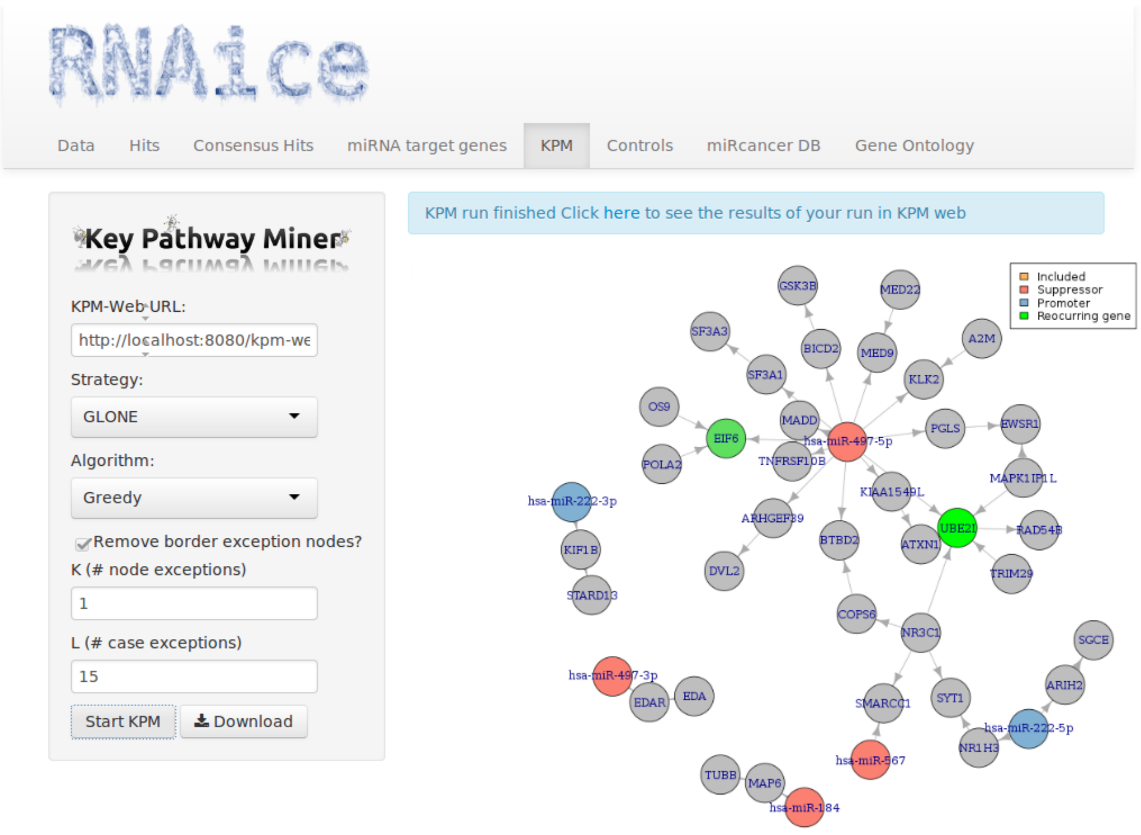
At the time of writing, no other enrichment tool included a built-in robustness or validation analysis. The addition of perturbations is important when establishing how solid the results are, when obtained from running KPM. With the implementation of this, it is possible to show how much results overlap when network graphs have been perturbed to certain degrees.

7.4.1 Integration of KPM into RNAice

RNAice is a R shiny based tool currently developed by Markus List at the NanoCAN Center (unpublished work). Its main purpose is to provide a platform for the interactive analysis of data from high-throughput screens, where experimental data derived from a large number of molecular experiments has to be analyzed. One use case of RNAice concerns the screening of microRNA (miRNA) inhibiting or mimicking molecules for their effect on the growth of various cell lines [1]. miRNAs are 19 to 25 nucleotide long small non-coding RNAs that control the expression levels of hundreds of genes.

Since the effect of miRNAs can only be explained by knowing the genes they are targeting, experimental evidence is a must. Currently, such evidence is mostly absent, even though target prediction algorithms are known. This is due to the fact that algorithms typically predict hundreds of target genes, which questions the precision of the algorithms. In order to learn which of the target genes are likely to explain an observed phenotype a better approach is needed. The hypothesis in RNAice is that by incorporating network information into the analysis, the list of target genes, responsible for change in cell growth, can be narrowed down significantly. The reasoning behind this hypothesis is based on the presumption that some miRNAs show an effect on cell growth target neighbouring genes in a network. If this is the case, then an enrichment tool like KPM should be able to find subgraphs that are enriched with those genes.

A proof-of-principle study of this hypothesis became available after integrating RNAice with KPM through the KPM web service (Section 6.4.3). RNAice generates an indicator matrix and utilizes the asynchronous execution model (Figure 21), allowing RNAice to retain its responsiveness for the user, while KPM runs on the server. A screenshot of executing KPM via RNAice is shown on figure 25.



KPM run finished Click here to see the results of your run in KPM web

KPM-Web URL: http://localhost:8080/kpm-we

Strategy: GLONE

Algorithm: Greedy

✓ Remove border exception nodes?

K (# node exceptions): 1

L (# case exceptions): 15

Figure 25: Screenshot of KPM integration with RNAice.

7.5 Updated requirements table

With the successful implementations of the extensions to KPM, it has been possible to satisfy the needs uncovered through the requirements analysis (Section 2). The updated requirements table can be found on Table 8.

	R1	R2	R3	R4	R5	R6	R7	R8
HotNet	✓	✗	✓	✓	✓	✗	✗	✗
MATISSE	✓	✗	✓	✗	✗	✗	✗	✗
jActiveModules	✓	✓	✗	✗	✗	✗	✗	✗
KPM 4.0	✓	✓	✓	✗	✗	✗	✓	✗
KPM 4.5	✓	✓	✓	✓	✓	✓	✓	✓

Table 8: Requirements table, updated with additions of this thesis. This includes a web service (R4, support integration), an web application (R5, web access), perturbation algorithms (R6, network perturbation), and result charts/graphs (R8, visualization of summary statistics).

8 Discussion

Before this thesis, KPM was already a stable and very effective enrichment tool. The uncovered needs found in the requirements analysis established that KPM could nevertheless be extended with novel features important to the end users. By providing a web platform to communicate and run KPM, the users can now access the enrichment tool through a browser. The web platform doubles as both a web application (R5) for the end user, and as a web service (R4) for other application to utilize. The network perturbation algorithms (R6) that were implemented, enabled robustness analyses to be performed. Additionally, charts for visualizing summary statistics (R8) were implemented. This helps the user in gaining an advantageous overview of the results. The updated requirements table is shown in Table 8, and it is clear that the extensions have increased usability by fulfilling all of the requirements that were still uncovered.

The purpose of this thesis was partly to analyze the current state of the enrichment tools available, and compare them to the needs, and requirements, of a biologist, or other types of bio-medical researchers working with OMICS data. Because the purpose of the analyses were to uncover needs that had not been addressed, there was no comparison between the effectiveness/performance of their respective implementations.

The results gathered through the requirements analysis (Section 2) brought valuable knowledge about where improvements of KPM should be made. While HotNet is available online, and therefore platform independent and possibly support integration with interfaces, it falls short in missing integration with Cytoscape and use of multiple datasets. The same can be said for MATISSE and jActiveModules. Neither of those support multiple datasets and in fact only KPM supports it, as far as the author is aware. jActiveModules seem to be available only as a Cytoscape, and thereby platform independent, but it is missing standalone and web access features. For all of the tools, there is room for improvements. KPM in its current version is lacking web access, support for integration and does not have visualization of statistical results.

The implementation of solutions to the uncovered needs, or requirements, has been the main focus of this thesis. Extracting the KPM to a library has been supported by design attributes such as maintainability and extensibility. It is maintainable because changes to the KPM algorithms in future versions only need to be implemented in one place, and since most objects from the library are accessed through interfaces, the impact is at a minimum. The latter argument also holds for either of the other two design attributes. Extensibility refers to the ability to extend KPM, such as a new network visualization tool. For both of these design attributes, the implementation of a library has increased both of their values. Additionally, KPM now facilitates easy integration, which refers to the ability to integrate an existing application with KPM.

Testing the robustness analysis on real data demonstrated to what extent the results found in a typical analysis are robust and stable upon perturbation of the networks. The validation analysis tests using a gold standard for the given test data could be used to verify that even with greater perturbation percentage, KPM still managed to find a significant number of gold standard nodes. Without such measurements of robustness and validation, the reliability of the results cannot be assessed and remain unclear. From the run-time tests it is clear that the perturbation and/or BEN-removal algorithms do not increase the overall run-time notably. Moreover, it was demonstrated how a simple R script (Section 6.5) can utilize the full power of KPM using very few lines of code. Finally, the web service integration test with RNAice proved that KPM can now indeed easily be integrated with external applications without any noticeable loss of functionality or performance.

A new intuitive web user interface was developed for KPM such that users seeking quick and uncomplicated access to results can use an instance of an online KPM web application, while users in need of a more powerful and custom experience can deploy their own KPM-web instance. This has the additional benefit that network enrichment analysis becomes instantly available to all members of the group. All that is needed is a server or virtual machine with java and a web application server installed. Moreover, this allows KPM to be deployed to cloud services like amazon EC2 easily. The latter has the advantage of being very scalable, enabling power users to execute huge KPM runs. This is especially beneficial since KPM supports multi-threading.

9 Conclusion

Despite the fact that highly effective enrichment tools are available to users, they are still not fulfilling a satisfactorily large amount of the needs of the users. In conclusion, there is a need for a more comprehensive enrichment tool that can be used across multiple platforms. A requirement analysis showed that there was a need for a web application that could relieve the users from having to install third-party applications, and focus on their research. Moreover, a need for a web service that can be integrated with external applications to utilize all features of KPM in a platform agnostic web interface. The newly developed KPM web application doubles as a web service, which has already proven useful in the integration of RNAice, a platform for the interactive analysis of data from high-throughput screens. The web application exposes the power of KPM to the users in an intuitive and simplified way. In this way, it is now possible to run KPM from a dedicated powerful server, using your own data, and see both pathways, charts, and statistics in one place without having to install third party software.

Additionally, by contributing robustness and validation analysis to KPM, the reliability of the results could be evaluated with ease. These evaluations are supported by charts, which can be obtained from any platform running KPM, creating a common look and feel for the results of KPM. The extensions helped KPM evolve to a more comprehensive enrichment tool, as well as an online platform, reachable by users and developers alike.

10 Outlook

By choosing a technology for implementation, the pitfalls of that technology will have to be handled properly. In this thesis, Grails and Hibernate has been used, and with it, special cases had to be handled. The first issue that was uncovered was with Hibernate, which is the technology for saving object in the web application. Because of the way Hibernate saves objects, it has trouble saving a lot of objects in the same transaction. For now, this issue was overcome by storing the list of elements in a CSV formatted string in the database. This cut saving time down to a fraction of what it was before. Another method for resolving the issue would be to produce and execute SQL code bundled in a transaction directly, as this is likely to be significantly faster.

Since KPM notifies the user via a task monitor, and the web application uses a database object to save these notifications, a lot of updates occur on the same object within a very short period of time. This results in Hibernate not flushing the updates for the user to see. The flushing occurs once the thread on which the object was first changed has terminated. The way to overcome this, is to make sure updates are executed on a new short-lived thread (Section 6.4.1).

While the implemented features to KPM works as intended, more relevant features can be added to the web application. One such feature could be to integrate the graph library with large network graph providers, like BioGrid. The advantages of continuously updating the graph library are clear, for instance the most relevant and up-to-date networks will always be available to the user.

Another refinement to the web application would be an expansion of the users management of runs, results and general settings. The ability to properly organize and group the runs and results, would enhance the user-friendliness and usability. In addition to this, an admin panel, exposing site-wide settings would also be beneficial to the administrators.

In addition to the graph perturbation techniques that were implemented and used as a part of the robustness and validation analysis, dataset perturbation techniques were also implemented in KPM core. While these perturbations work as intended, a user interface has yet to be defined, before they can be utilized properly.

Contact has been made with the Navigator development team [7] to include KPM as a plugin. Navigator is a network analysis and graphing software much like Cytoscape. As a result of the work performed in this thesis, porting the functionality to this new platform is simplified.

Finally, it is likely that the new interfaces and access strategies implemented in this thesis will help to further spread the use of KPM as an integrated part of third party tools. In addition, the newly developed KPM web application is expected to increase the visibility of the project through its easy access for a large audience of bio-medical researchers.

References

[1] Mechanisms of post-transcriptional regulation by microRNAs: are the answers in sight? *Nature reviews. Genetics*, 9(2):102–14, Feb. 2008. ISSN 1471-0064. URL `http://www.ncbi.nlm.nih.gov/pubmed/18197166`.

[2] N. Alcaraz. KeyPathwayMiner – Detecting Case-specific Biological Pathways by Using Expression Data. 2012.

[3] N. Alcaraz, T. Friedrich, T. Kötzing, A. Krohmer, J. Müller, J. Pauling, and J. Baumbach. Efficient key pathway mining: combining networks and OMICS data. *Integrative Biology*, 4:756, 2012. ISSN 1757-9694. doi: 10.1039/c2ib00133k.

[4] N. Alcaraz, J. Pauling, R. Batra, E. Barbosa, A. Junge, A. G. L. Christensen, V. Azevedo, H. J. Ditzel, and J. Baumbach. KeyPathwayMiner 4.0: condition-specific pathway analysis by combining multiple omics studies and networks with Cytoscape. *BMC systems biology*, 8:99, Jan. 2014. ISSN 1752-0509. doi: 10.1186/s12918-014-0099-x. URL `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4236746&tool=pmcentrez&rendertype=abstract`.

[5] J. Baumbach, T. Friedrich, T. Kötzing, A. Krohmer, J. Müller, and J. Pauling. Efficient algorithms for extracting biological key pathways with global constraints. *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference - GECCO '12*, page 169, 2012. doi: 10.1145/2330163.2330188. URL `http://dl.acm.org/citation.cfm?id=2330163.2330188`.

[6] M. Bostock. JavaScript library D3.js, 2011. URL `http://d3js.org`.

[7] K. R. Brown, D. Otasek, M. Ali, M. J. McGuffin, W. Xie, B. Devani, I. L. van Toch, and I. Jurisica. NAViGaTOR: Network analysis, visualization and graphing Toronto. *Bioinformatics*, 25(24):3327–3329, 2009.

[8] T. Cancer and G. Atlas. Comprehensive molecular characterization of human colon and rectal cancer. *Nature*, 487(7407):330–7, 2012. ISSN 1476-4687. doi: 10.1038/nature11252. URL `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3401966&tool=pmcentrez&rendertype=abstract`.

[9] A. Couture-Beil. Package 'RJSON', 2014. URL `http://cran.r-project.org/web/packages/rjson/index.html`.

[10] M. Dorigo and M. Birattari. Ant colony optimization. In C. Sammut and G. Webb, editors, *Encyclopedia of Machine Learning*, pages 36–39. Springer US, 2010. ISBN 978-0-387-30768-8. doi: 10.1007/978-0-387-30164-8_22. URL `http://dx.doi.org/10.1007/978-0-387-30164-8_22`.

[11] J. Dreo. Shortest path find by an ant colony, 2006. URL `http://en.wikipedia.org/wiki/File:Aco_branches.svg`.

[12] T. Duncan. Package 'RCurl', 2014. URL `http://cran.r-project.org/web/packages/RCurl/index.html`.

[13] T. Gene and O. Consortium. Gene Ontology - tool for the. 25(may):25–29, 2000.

[14] D. Gilbert. Jfreechart, java chart library, 2000. URL `http://www.jfree.org/jfreechart/`.

[15] T. Ideker, O. Ozier, B. Schwikowski, and F. Andrew. Discovering regulatory and signalling circuits in molecular interaction networks. *Bioinformatics (Oxford, England)*, 18: 233–240, 2002.

[16] A. Jacomy. JavaScript library Sigma.js, 2012. URL `http://sigmajs.org`.

[17] A. R. Joyce and B. O. Palsson. The model organism as a system: integrating 'omics' data sets. *Nature reviews. Molecular cell biology*, 7(March):198–210, 2006. ISSN 1471-0072. doi: 10.1038/nrm1857.

[18] M. Kanehisa and S. Goto. KEGG : Kyoto Encyclopedia of Genes and Genomes. 28(1): 27–30, 2000.

[19] S. Maslov and K. Sneppen. Specificity and stability in topology of protein networks. *Science (New York, N.Y.)*, 296(5569):910–913, 2002.

[20] J. Ostell and J. McEntyre. The NCBI Handbook, 2007. URL `www.ncbi.nlm.nih.gov/books/NBK21101/?report=printable`.

[21] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2013. URL `http://www.R-project.org/`.

[22] M. E. Smoot, K. Ono, J. Ruscheinski, P.-L. Wang, and T. Ideker. Cytoscape 2.8: new features for data integration and network visualization. *Bioinformatics (Oxford, England)*, 27(3):431–2, Feb. 2011. ISSN 1367-4811. doi: 10.1093/bioinformatics/btq675. URL `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3031041&tool=pmcentrez&rendertype=abstract`.

[23] I. Ulitsky and R. Shamir. Identification of functional modules using network topology and high-throughput data. *BMC systems biology*, 1:8, Jan. 2007. ISSN 1752-0509. doi: 10.1186/1752-0509-1-8. URL `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1839897&tool=pmcentrez&rendertype=abstract`.

[24] I. Ulitsky, R. Karp, and R. Shamir. Detecting disease-specific dysregulated pathways via analysis of clinical expression profiles. *Research in Computational Molecular ...*, pages 347–359, 2008. URL `http://link.springer.com/chapter/10.1007/978-3-540-78839-3_30`.

[25] F. Vandin, E. Upfal, and B. J. Raphael. Algorithms for detecting significantly mutated pathways in cancer. *Journal of computational biology : a journal of computational molecular cell biology*, 18(3):507–22, Mar. 2011. ISSN 1557-8666. doi: 10.1089/cmb.2010.0265. URL `http://www.ncbi.nlm.nih.gov/pubmed/21385051`.

[26] S. White. Jung - java universal network/graph framework. 2003. URL `http://jung.sourceforge.net`.

A Full requirements table

Requirements	
Platform independent	R1
Cytoscape	R2
Standalone	R3
Support integration	R4
Web access	R5
Network perturbation	R6
Multiple datasets	R7
Visualization of summary statistics	R8

Table 9: This table lists the requirements identified as important for state of the art network enrichment tools.

	R1	R2	R3	R4	R5	R6	R7	R8
BioNET	✓	✗	✓	✓	✗	✗	✗	✗
GXNA	✓	✗	✓	✓	✗	✗	✗	✗
MEMo	✓	✗	✓		✗	✗	✗	✗
PARADIGM	✓	✗	✓	✓	✗	✗	✗	✗
ResponseNET	✓	✗	✓	✗	✓	✗	✗	✗
EXPANDER	✓	✗	✓	✗	✗	✗	✗	✗
HotNet	✓	✗	✓	✓	✓	✗	✗	✗
MATISSE	✓	✗	✓	✗	✗	✗	✗	✗
jActiveModules	✓	✓	✗	✗	✗	✗	✗	✗
KPM 4.0	✓	✓	✓	✗	✗	✗	✓	✗

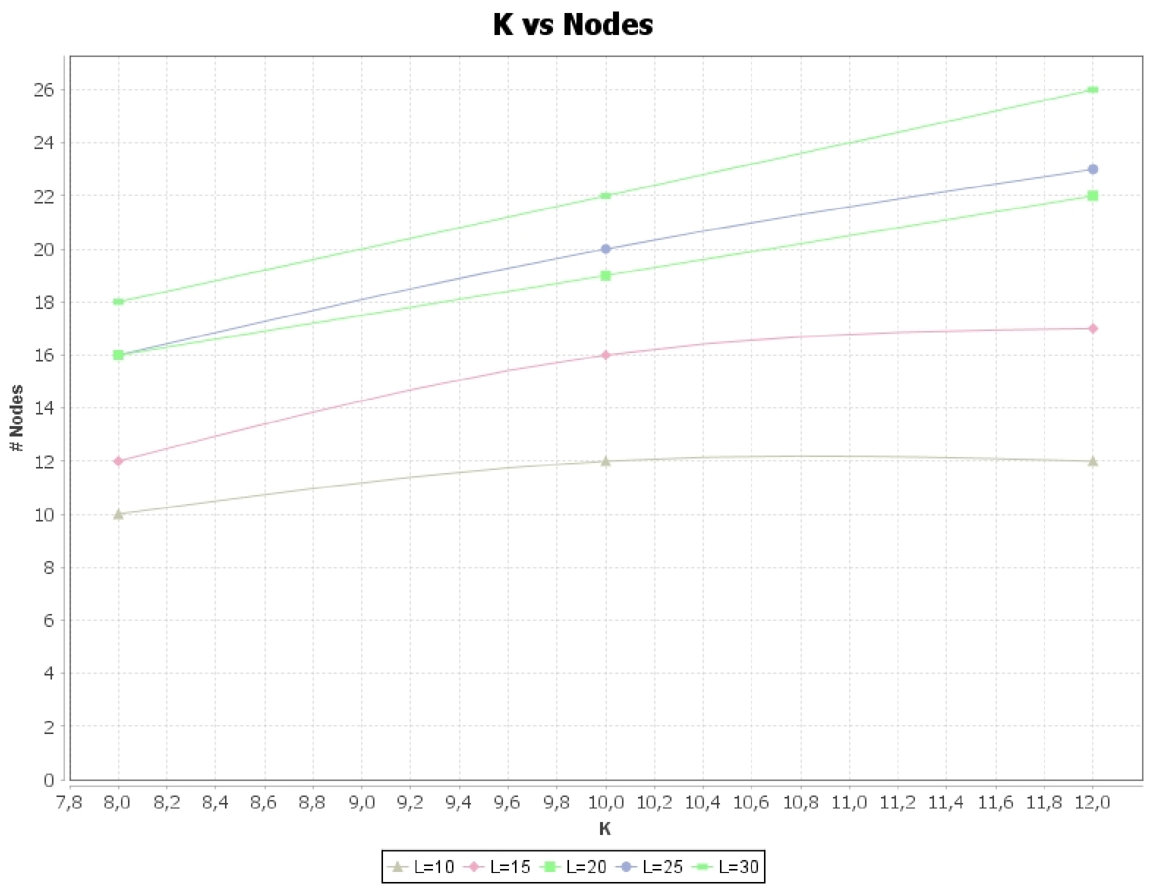
Table 10: This table shows how all of the state of the art tools fulfilled the requirements uncovered through the requirements analysis in Section 3. This is before the addition of this thesis.

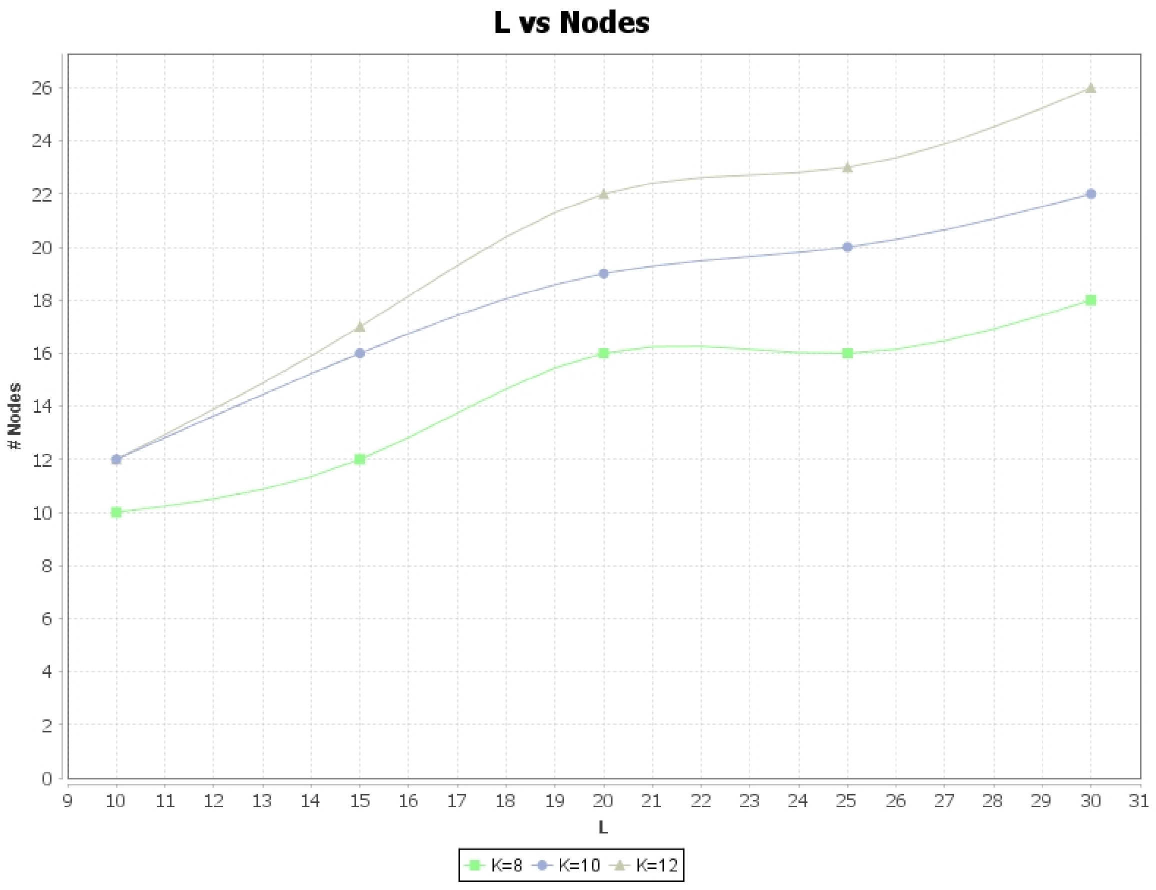
B Results output

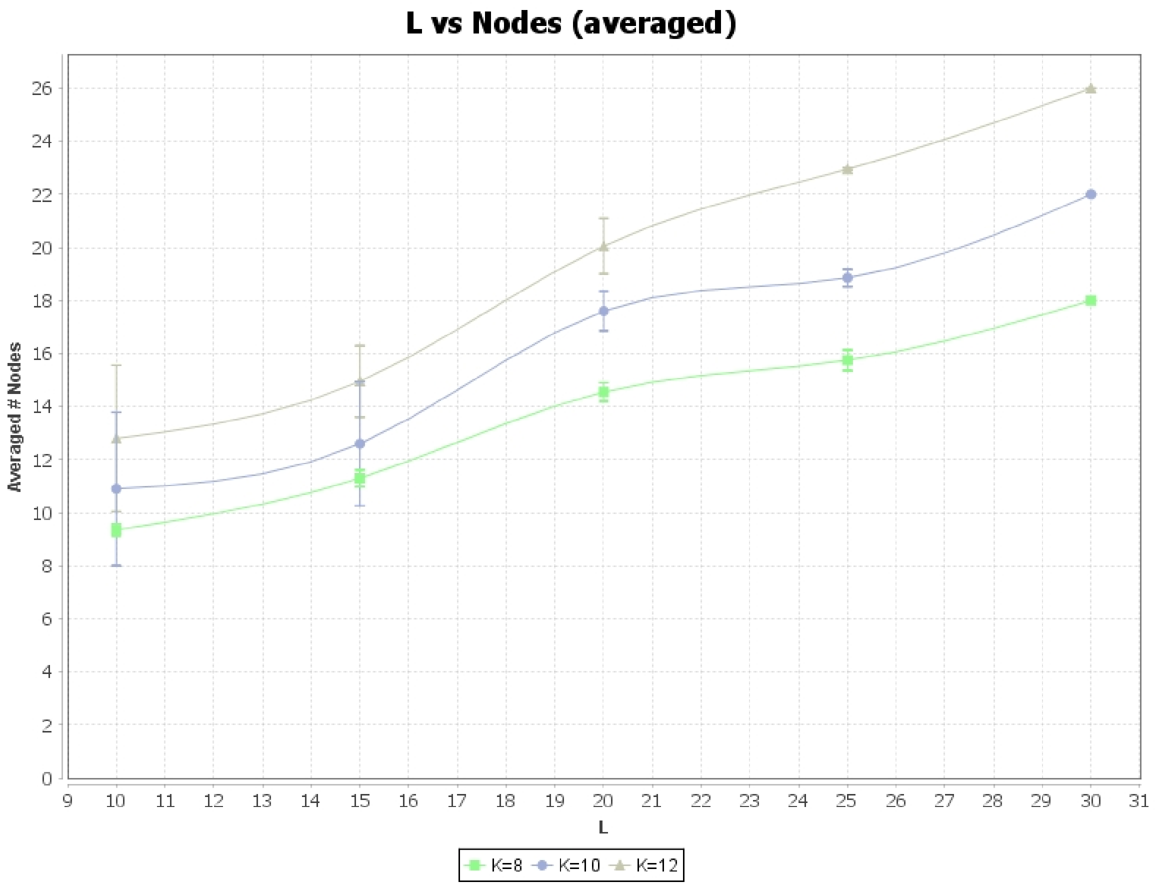
B.1 Robustness kpm-standalone parameters

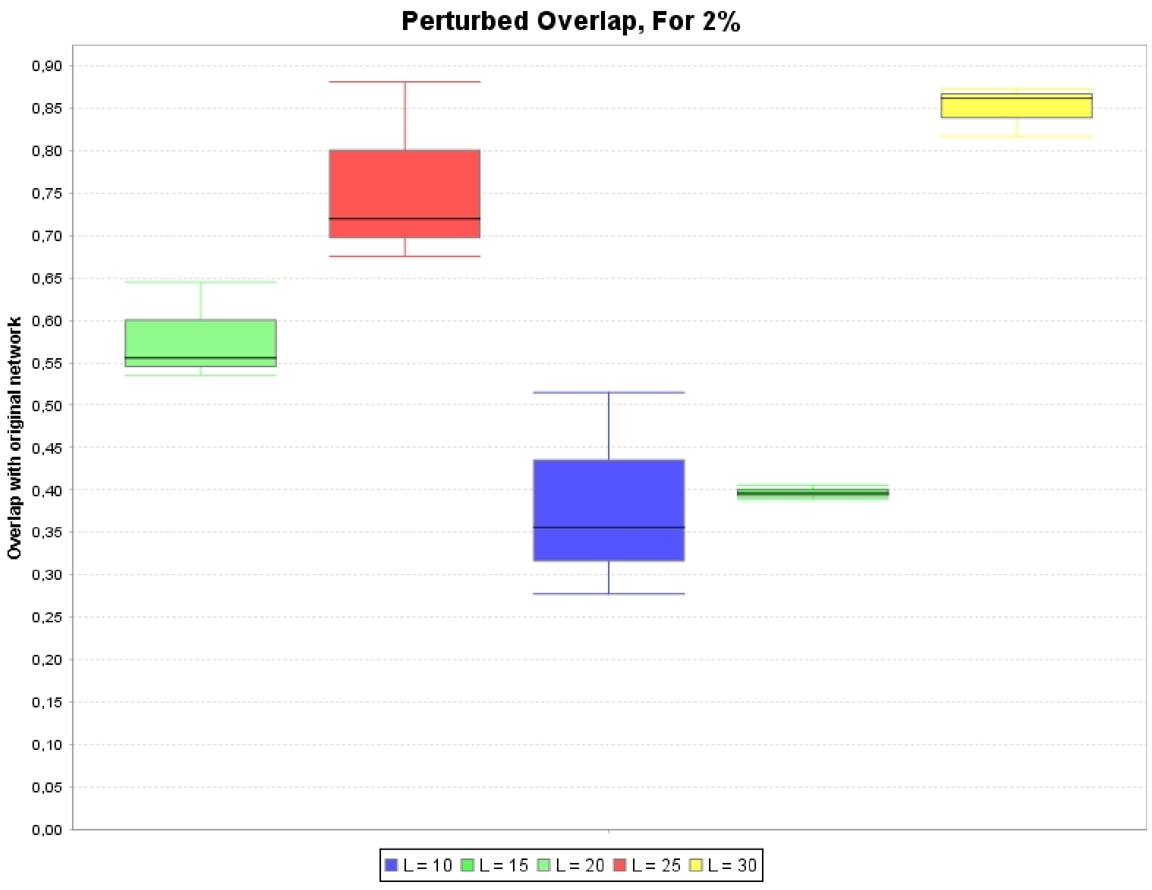
```
-algo=GREEDY
-strategy=INES
-graphFile=data/graph-ulitsky-entrez.sif
-matrix1=data/COAD-EXP-UP-p0.05.txt
-batch
-K_batch=6,2,14
-L1_batch=10,5,40
-resultsDir=results
-perturbation=10,1,12,1
-perturbation_technique=noderemove
```

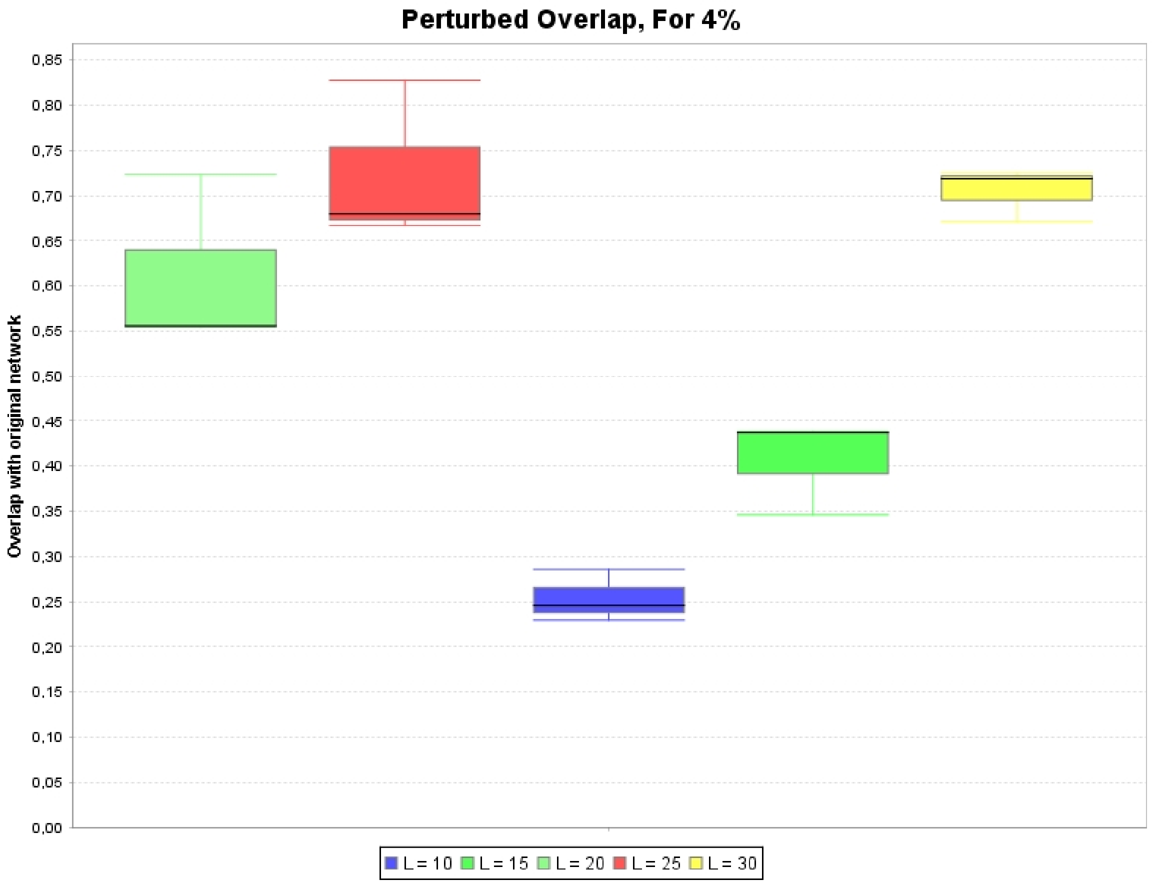
B.2 Robustness analysis charts

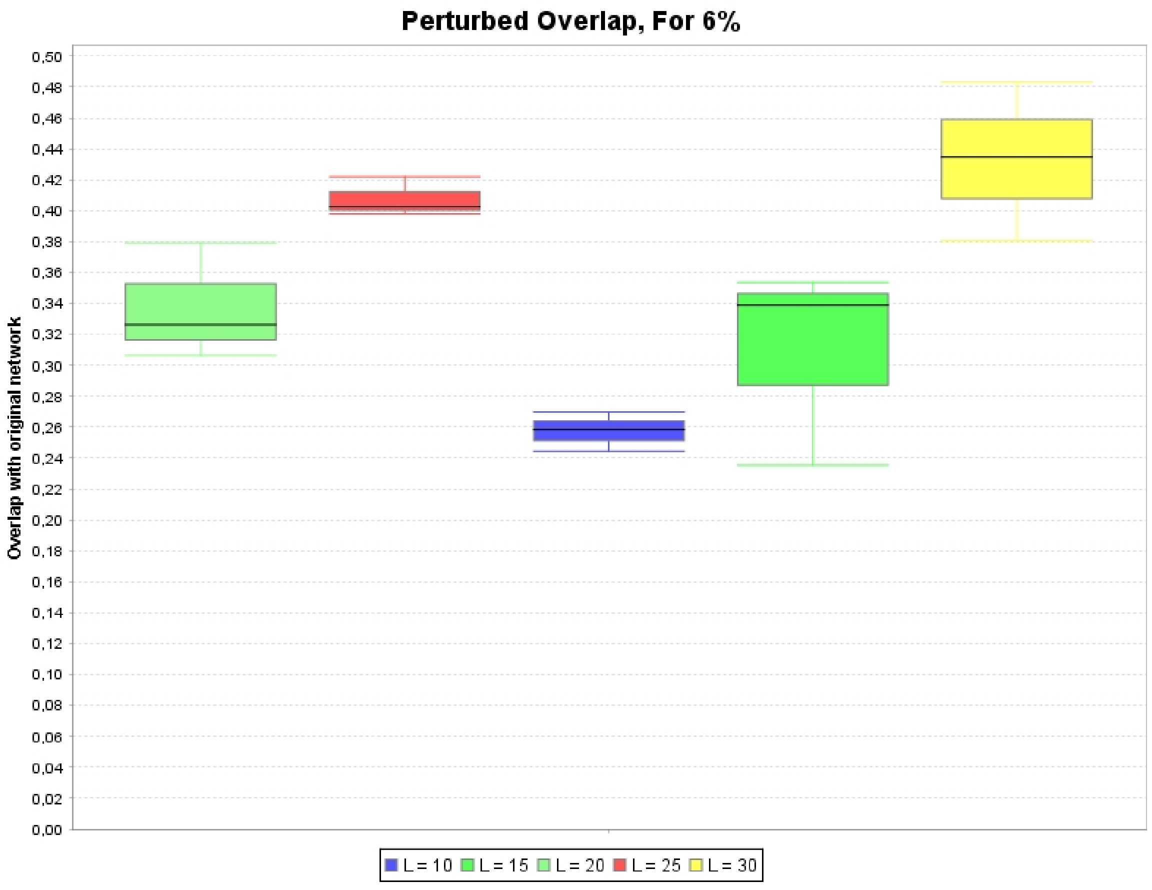


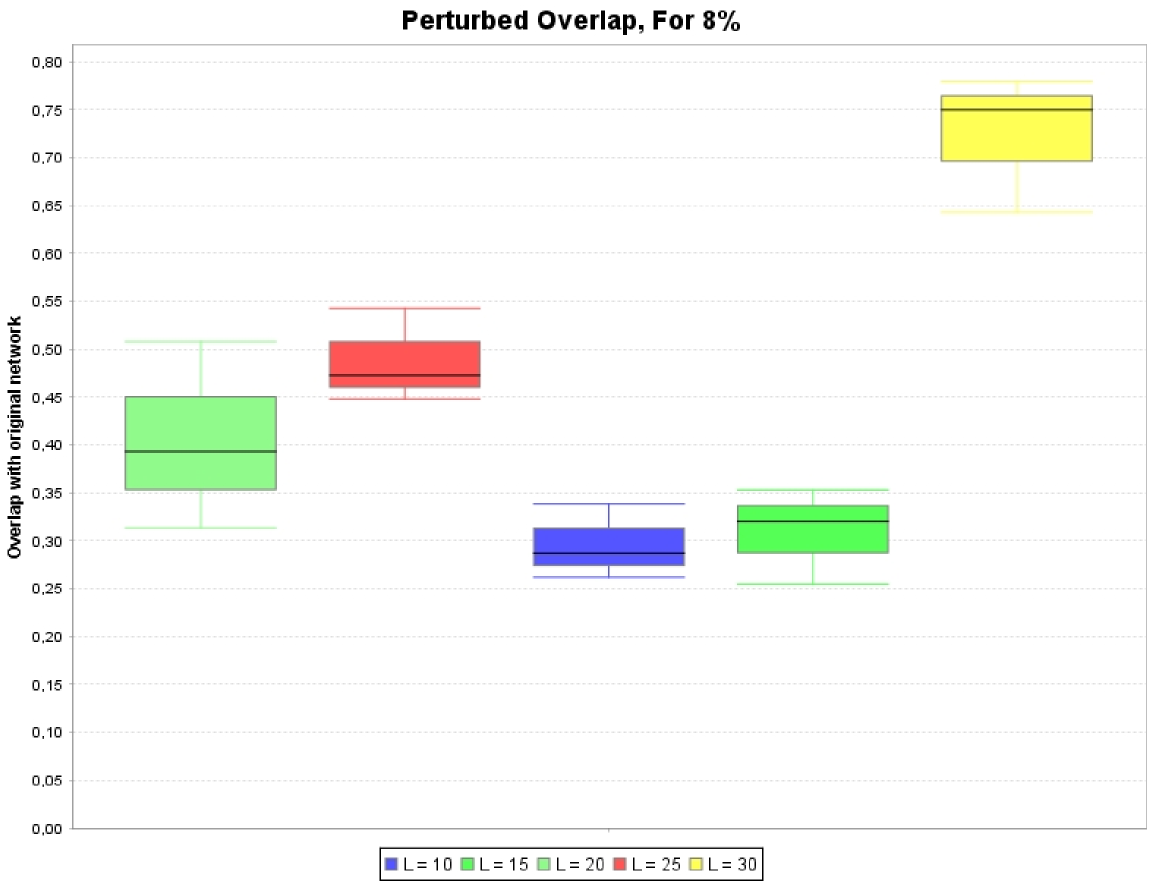

L vs Nodes
# Nodes
L
K=8
K=10
K=12

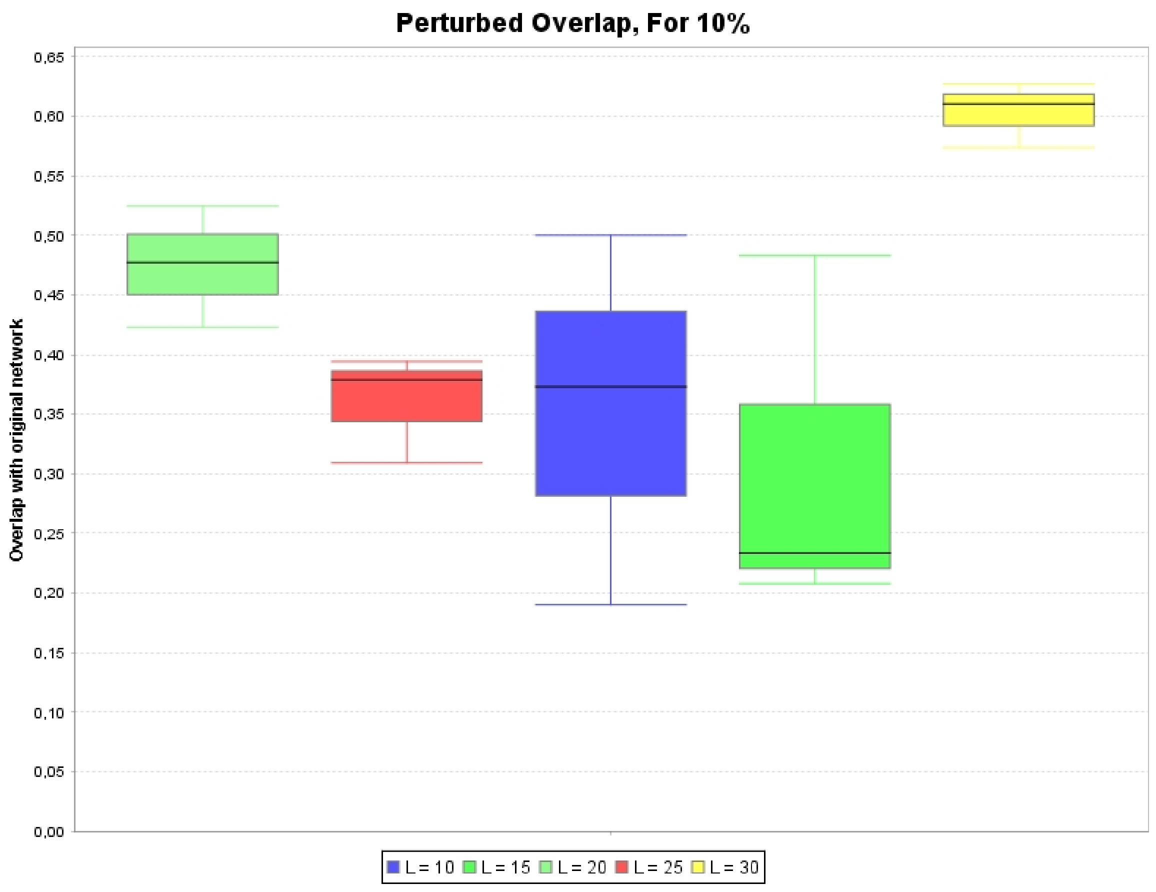

L vs Nodes (averaged)
Averaged # Nodes
L
K=8
K=10
K=12


Perturbed Overlap, For 2%
Overlap with original network
0,90
0,85
0,80
0,75
0,70
0,65
0,60
0,55
0,50
0,45
0,40
0,35
0,30
0,25
0,20
0,15
0,10
0,05
0,00
🟦 L = 10 🟩 L = 15 🟩 L = 20 🟥 L = 25 🟨 L = 30

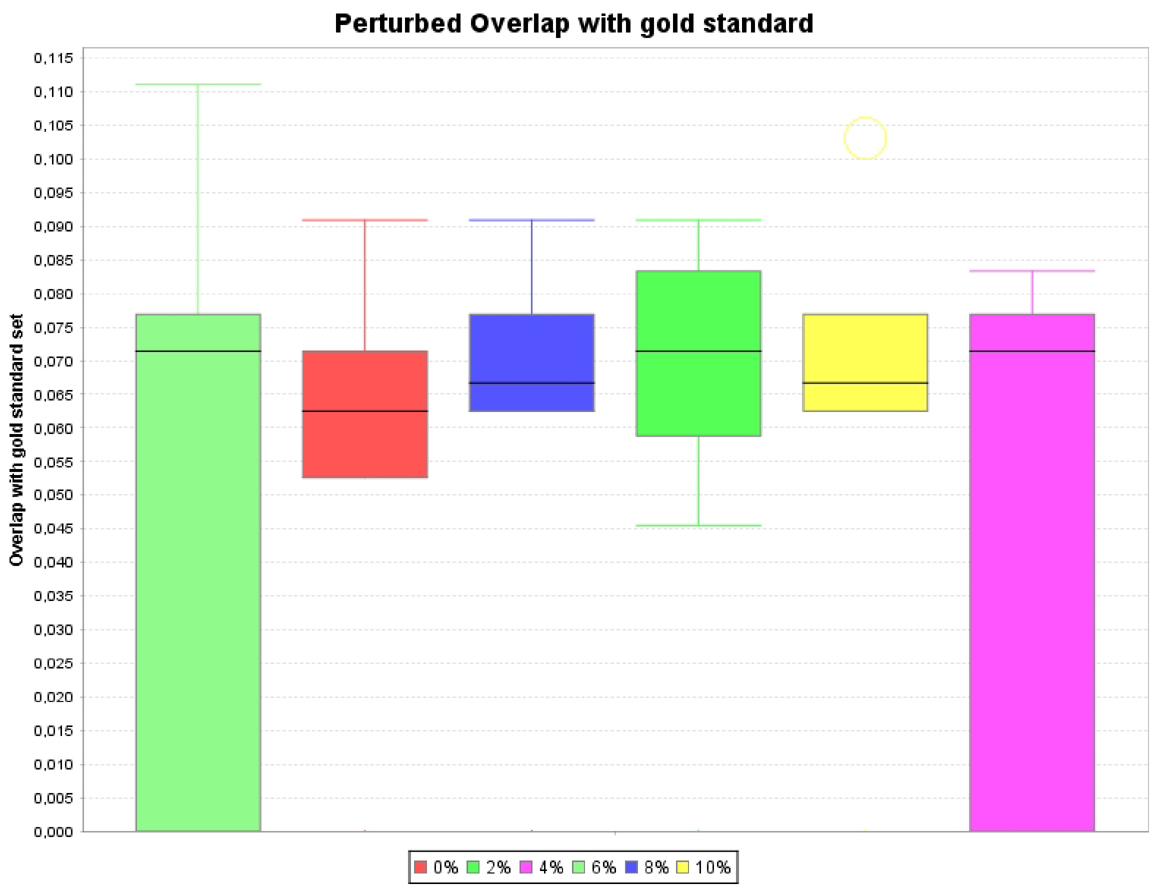

Perturbed Overlap, For 4%
Overlap with original network
0,85
0,80
0,75
0,70
0,65
0,60
0,55
0,50
0,45
0,40
0,35
0,30
0,25
0,20
0,15
0,10
0,05
0,00
🟦 L = 10 🟩 L = 15 🟩 L = 20 🟥 L = 25 🟨 L = 30


Perturbed Overlap, For 6%
Overlap with original network
0,50
0,48
0,46
0,44
0,42
0,40
0,38
0,36
0,34
0,32
0,30
0,28
0,26
0,24
0,22
0,20
0,18
0,16
0,14
0,12
0,10
0,08
0,06
0,04
0,02
0,00
🟦 L = 10 🟩 L = 15 🟩 L = 20 🟥 L = 25 🟨 L = 30


Perturbed Overlap, For 8%
Overlap with original network
0,80
0,75
0,70
0,65
0,60
0,55
0,50
0,45
0,40
0,35
0,30
0,25
0,20
0,15
0,10
0,05
0,00
🟦 L = 10 🟩 L = 15 🟩 L = 20 🟥 L = 25 🟨 L = 30


Perturbed Overlap, For 10%
Overlap with original network
0,65
0,60
0,55
0,50
0,45
0,40
0,35
0,30
0,25
0,20
0,15
0,10
0,05
0,00
L = 10
L = 15
L = 20
L = 25
L = 30

B.3 Validation analysis chart



B.4 Runtime test results

Batch run times	**Perturbation runtimes**
58115 ms	2678 ms
37233 ms	2238 ms
55131 ms	2276 ms
38585 ms	2248 ms
36317 ms	2156 ms
42968 ms	2149 ms
45521 ms	2264 ms
43861 ms	2516 ms
37922 ms	2194 ms
44316 ms	2204 ms
43996,9 ms avg	**2292,3 ms avg**

Table 11: Runtime results in milliseconds, test results for clean KPM batch run, with $k = 20$ and l ranging from 20 to 30, with a step of 10, and includes BEN-removal. The perturbation runtimes are the result of running the perturbation algorithm "node-remove" with 10% perturbation.

Test machine specifications
Windows 8.1 Pro
Intel Core i7-3770 CPU 3.40GHz
16 GB RAM

Table 12: This table shows the specifications for the machine on which all tests were performed

C Guide to KPM web application

This will be a step by step guide to use the new KPM web application.

Step 0 - choosing functionality

In addition to containing some standard datasets and graphs, KPM web application also enables the user to add their own. These will be strictly visible to only the user that is logged in, or the session to which the user is associated. The functionality to do so can be reached by clicking on either the "graphs" or "datasets" tab shown in Figure 26. The setup of the run parameters starts by clicking on the "run" tab.

It is also possible to sign up as a user. This is done via the "create account" link in the right side of the header. By signing up as a user, it is possible to retrieve old results, as they will be stored in a database, along with graphs and charts.

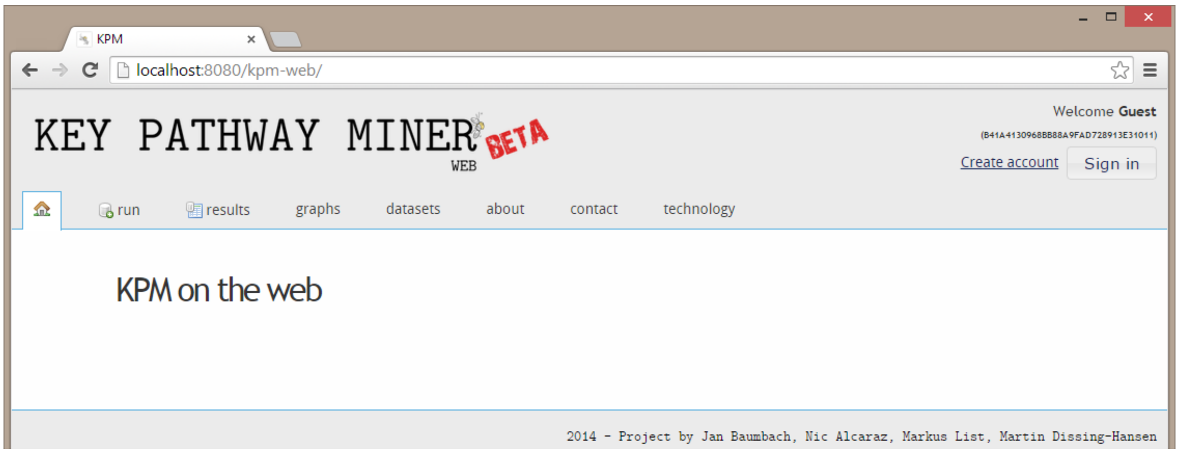
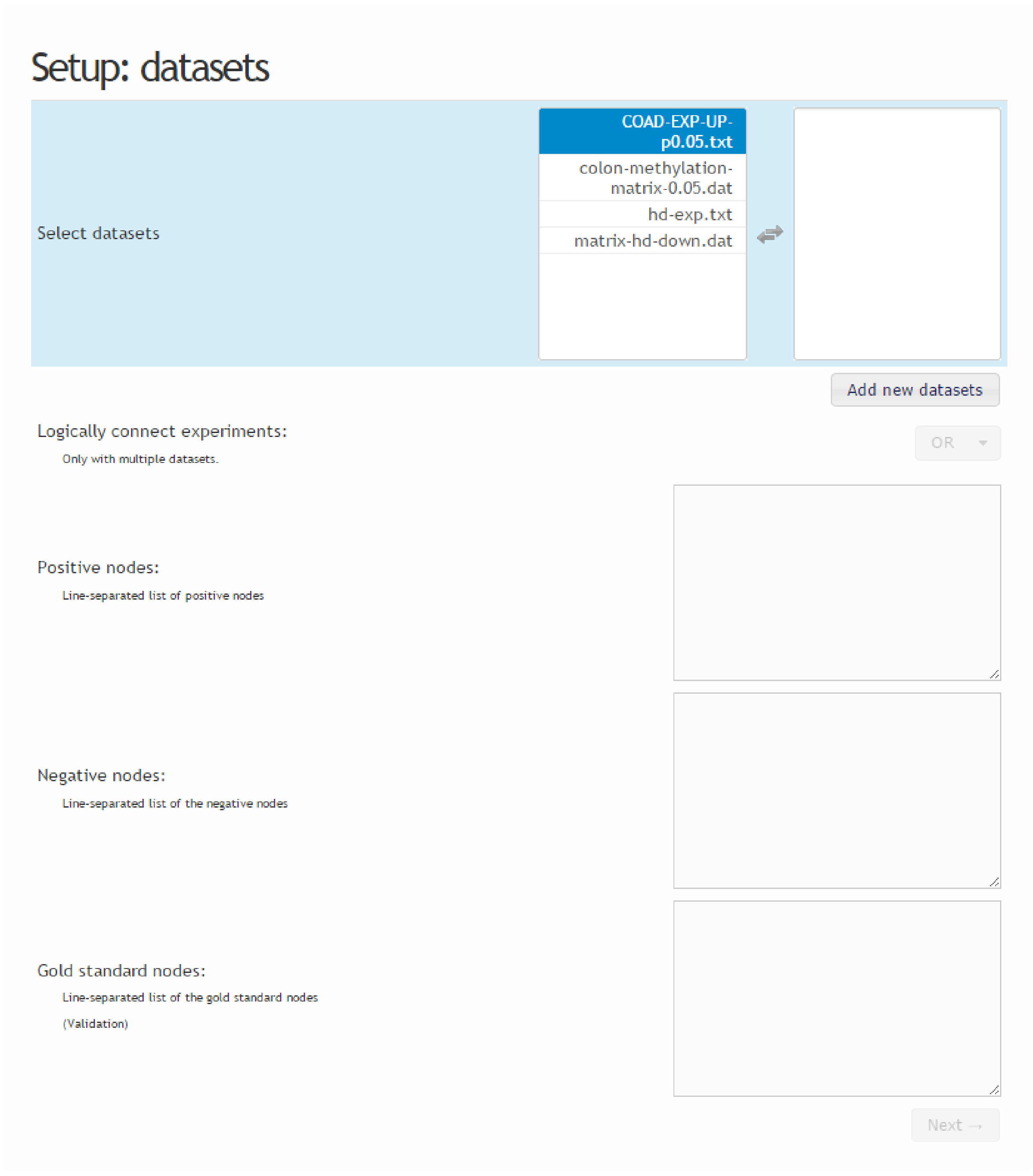


Figure 26: The header of the KPM web application.

Step 1 - datasets

In this step, the focus is on the datasets, and the nodes from these, as shown on Figure 27. The first option is what dataset(s) to use. It is possible to select multiple datasets, and connect them through a logical operators. It is also possible to be redirected to the page for adding new datasets, via the button "Add new datasets".

To add special nodes to either the positive or the negative nodes, it can be achieved by writing them, in a line-separated manner in the following two text-areas. Lastly is the setting for comparing with gold standard nodes, which is for validation analysis.



Setup: datasets

Select datasets

COAD-EXP-UP-p0.05.txt
colon-methylation-matrix-0.05.dat
hd-exp.txt
matrix-hd-down.dat

Logically connect experiments:
Only with multiple datasets.

Positive nodes:
Line-separated list of positive nodes

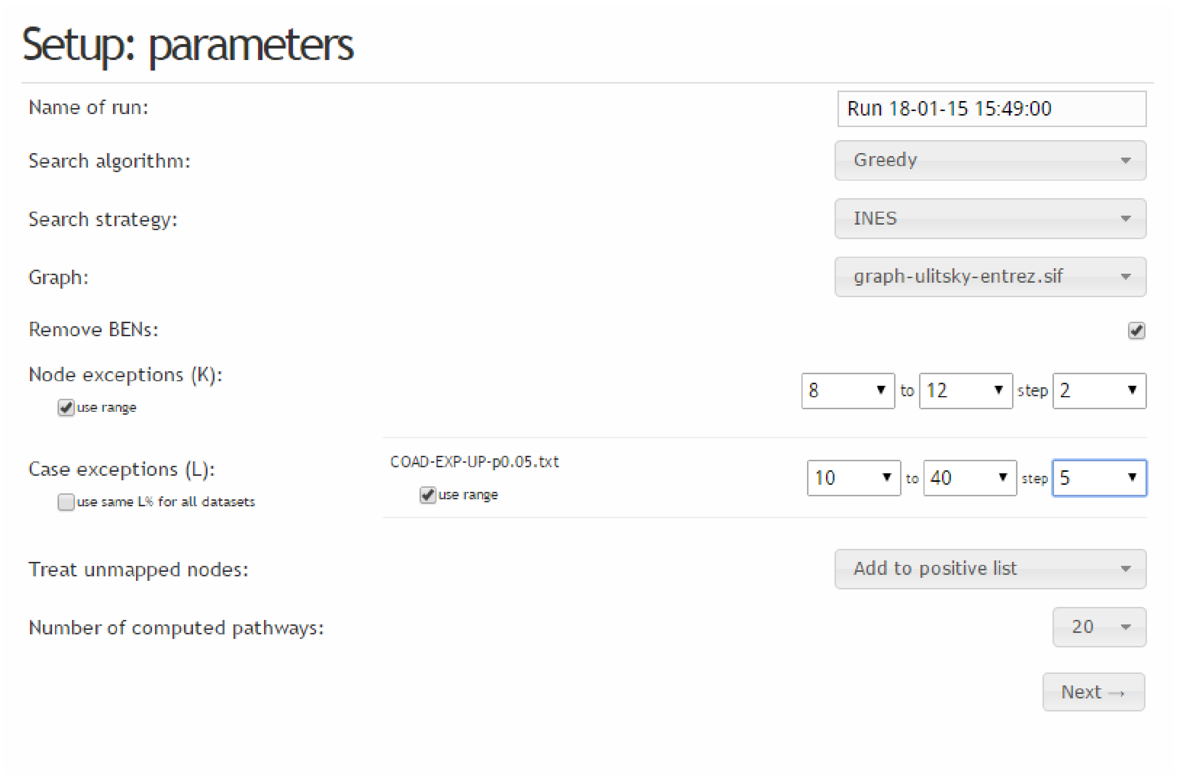
Negative nodes:
Line-separated list of the negative nodes

Gold standard nodes:
Line-separated list of the gold standard nodes
(Validation)

Figure 27: Setting up datasets and special nodes.

Step 2 - Parameters

In this step, the rest of the main parameters are set up. This is where it is possible to setup the name of the run, the graph, the search algorithm and strategy, computed pathways, how to treat unmapped nodes, and whether or not to remove BENs. It is also possible to choose both single value and ranges for node and case exceptions, exactly like in the Cytoscape app. Once the user has finished configuring the parameters, the "next" button can be clicked to get to the next step.



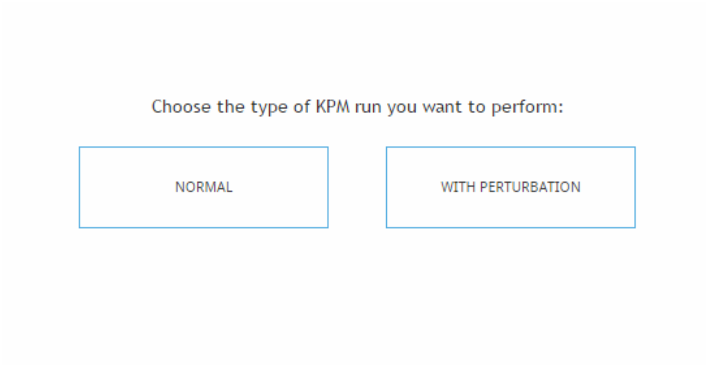
Setup: parameters

Name of run:	Run 18-01-15 15:49:00
Search algorithm:	Greedy
Search strategy:	INES
Graph:	graph-ulitsky-entrez.sif
Remove BENs:	✓
Node exceptions (K): ✓ use range	8 to 12 step 2
Case exceptions (L): use same L% for all datasets	COAD-EXP-UP-p0.05.txt ✓ use range 10 to 40 step 5
Treat unmapped nodes:	Add to positive list
Number of computed pathways:	20

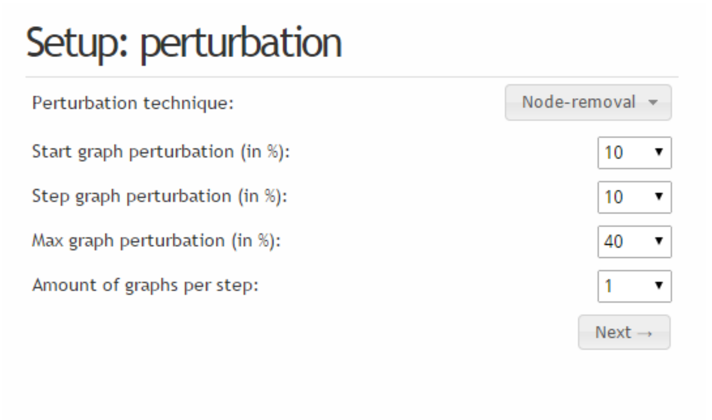
Figure 28: Setting up run parameters.

Step 3 - Run type

In this step, the user has to choose whether or not to run KPM normally, or with perturbation (robustness/validation run), as shown on Figure 29a. If "normal" is selected, the user proceeds to the next step. If the user selects "with perturbation", the user is presented with perturbation parameters to setup, shown on Figure 29b, which includes all the perturbations implemented. The user finishes the configuration and continues to next step.



(a)


Setup: perturbation

Perturbation technique: Node-removal

Start graph perturbation (in %): 10

Step graph perturbation (in %): 10

Max graph perturbation (in %): 40

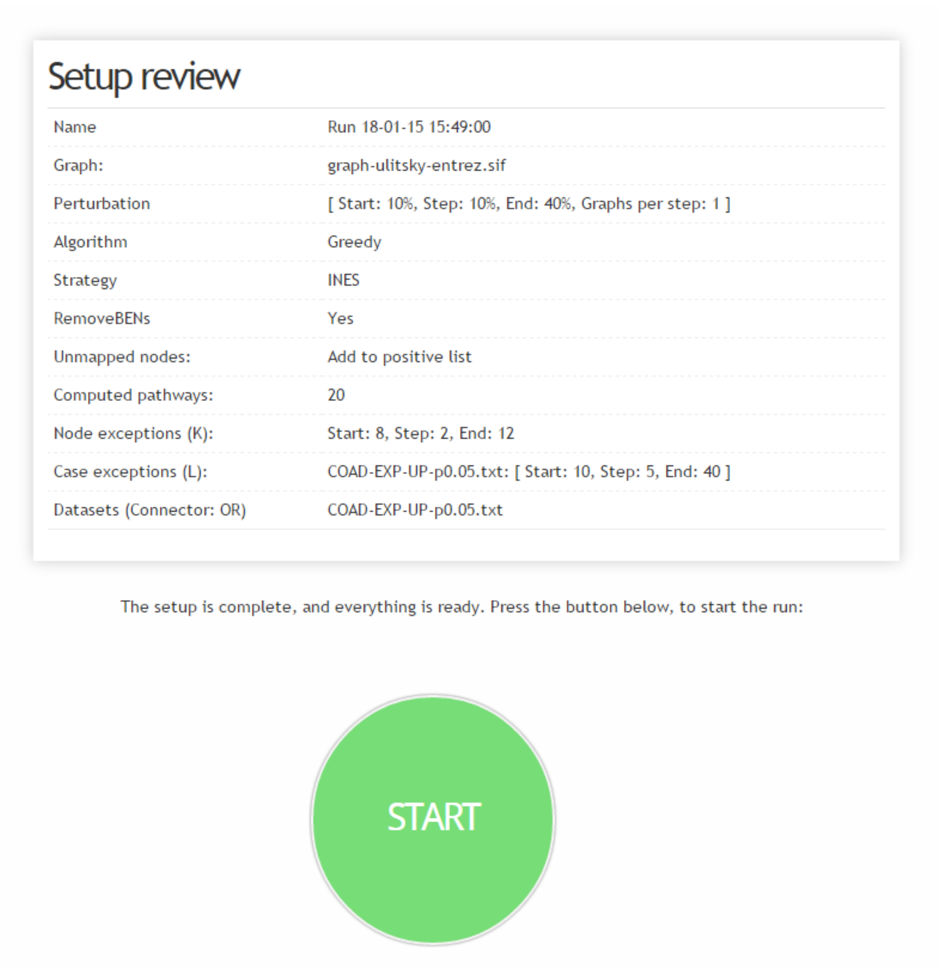
Amount of graphs per step: 1

(b)

Figure 29: Choosing the run type.

Step 4 - setup review

Once the rest of the configurations has completed, the setup can be reviewed from this step, shown on Figure 30. If the user wants to proceed with the run, then the big green "start" button is pressed. and the user is directed to the "results" tab.



Setup review

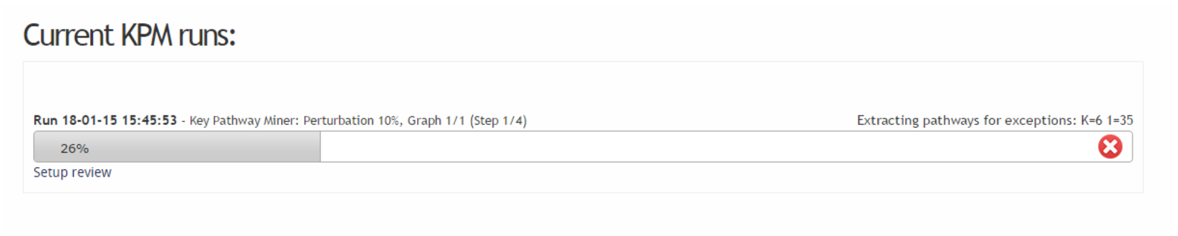
Name	Run 18-01-15 15:49:00
Graph:	graph-ulitsky-entrez.sif
Perturbation	[Start: 10%, Step: 10%, End: 40%, Graphs per step: 1]
Algorithm	Greedy
Strategy	INES
RemoveBENs	Yes
Unmapped nodes:	Add to positive list
Computed pathways:	20
Node exceptions (K):	Start: 8, Step: 2, End: 12
Case exceptions (L):	COAD-EXP-UP-p0.05.txt: [Start: 10, Step: 5, End: 40]
Datasets (Connector: OR)	COAD-EXP-UP-p0.05.txt

The setup is complete, and everything is ready. Press the button below, to start the run:

Figure 30: Reviewing settings.

Step 5 - current KPM runs

In the "results" tab redirects to a page where the user can see all currently running and completed quests in the form of a loading bar, as shown in Figure 31.



Current KPM runs:

Run 18-01-15 15:45:53 - Key Pathway Miner: Perturbation 10%, Graph 1/1 (Step 1/4)

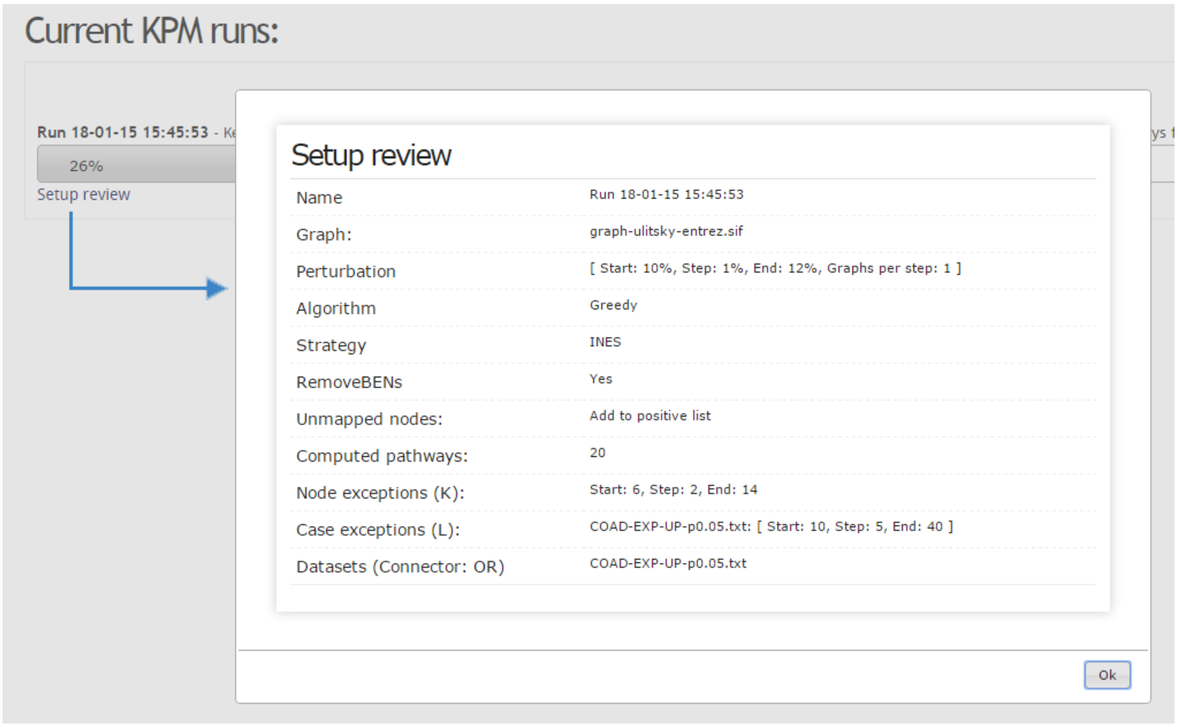
Extracting pathways for exceptions: K=6 l=35

26%

Setup review

Figure 31: Current KPM runs.

In case the user wants to review what the exact setup to a given run is, a click on the "setup review" will show the modal overlay visible on Figure 32.



Current KPM runs:

Run 18-01-15 15:45:53

26%

Setup review

Setup review

Name	Run 18-01-15 15:45:53
Graph:	graph-ulitsky-entrez.sif
Perturbation	[Start: 10%, Step: 1%, End: 12%, Graphs per step: 1]
Algorithm	Greedy
Strategy	INES
RemoveBENs	Yes
Unmapped nodes:	Add to positive list
Computed pathways:	20
Node exceptions (K):	Start: 6, Step: 2, End: 14
Case exceptions (L):	COAD-EXP-UP-p0.05.txt: [Start: 10, Step: 5, End: 40]
Datasets (Connector: OR)	COAD-EXP-UP-p0.05.txt

Figure 32: Current KPM runs, review of settings for a given run.

Step 6 - results

Once a run has completed, a "see results" button will appear underneath the loading bar (Figure 31) for that run. This will redirect the user to a page containing the results. The first that will meet the user is the list of charts that were generated as a part of the KPM run, as shown on Figure 33. Underneath this list of charts, the setup for the network graphs will appear as a set of sliders and three buttons. On Figure 34 this setup is shown. Once the user has chosen a suitable k/l/node set value, the "show graph" button can be pressed. This will end up in a graph like that on Figure 35. The "show union graph" button will fetch the union graph and show it as a colored graph, Figure 36, which also contains a color-meter, showing what each color mean in relation to the occurrence amount. If a node is clicked the page of that node on the NCBI website will be opened (Figure 37).

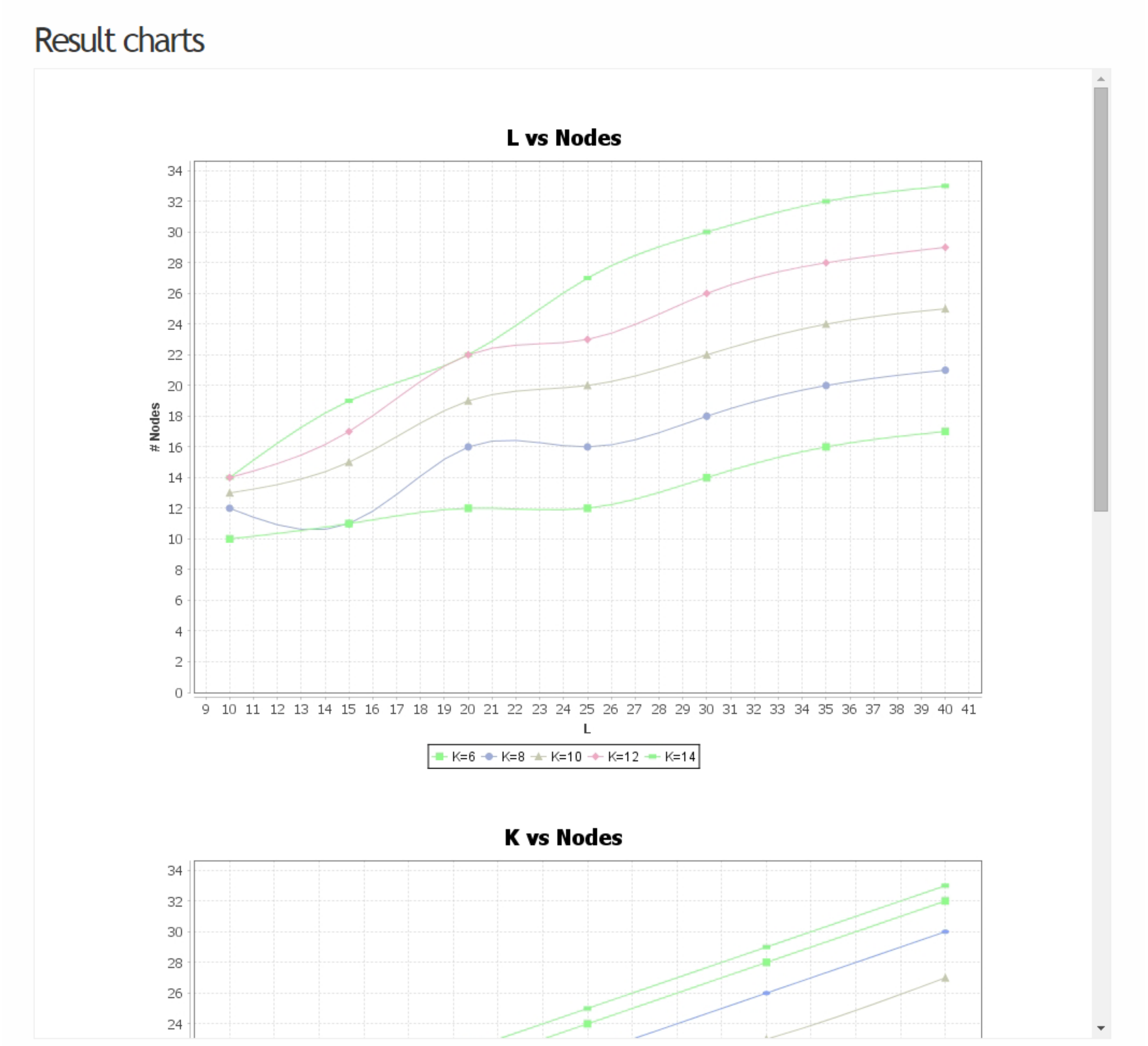
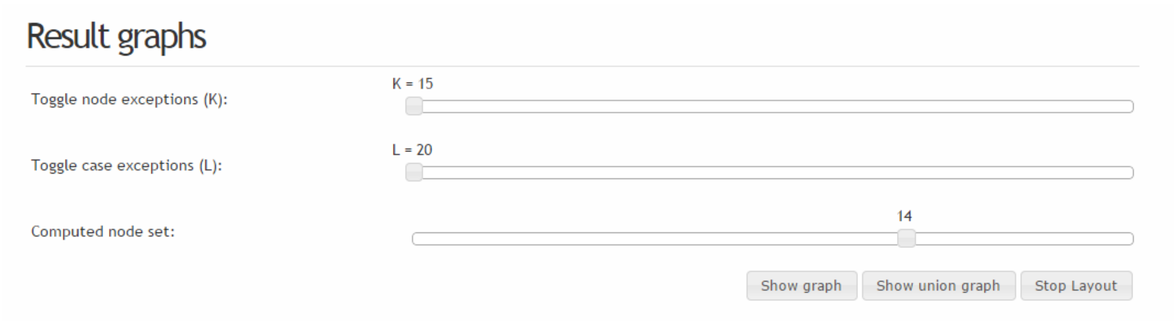


Figure 33: Result charts from a KPM run



Result graphs

Toggle node exceptions (K): K = 15

Toggle case exceptions (L): L = 20

Computed node set: 14

Figure 34: Network graphs of the results

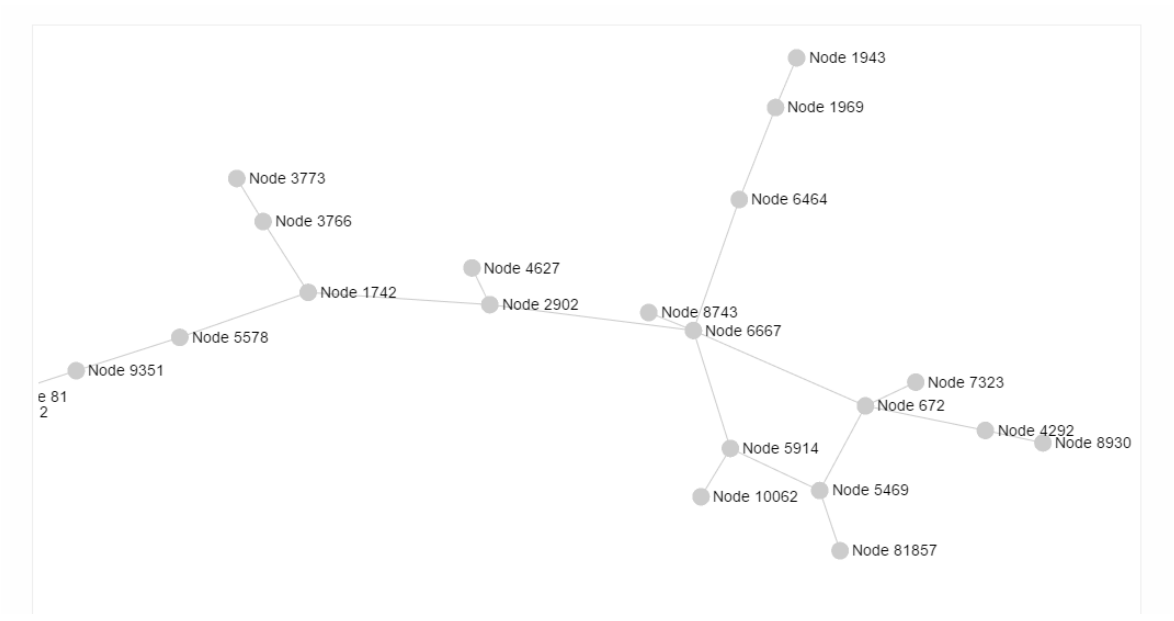


Figure 35: Filled network graph

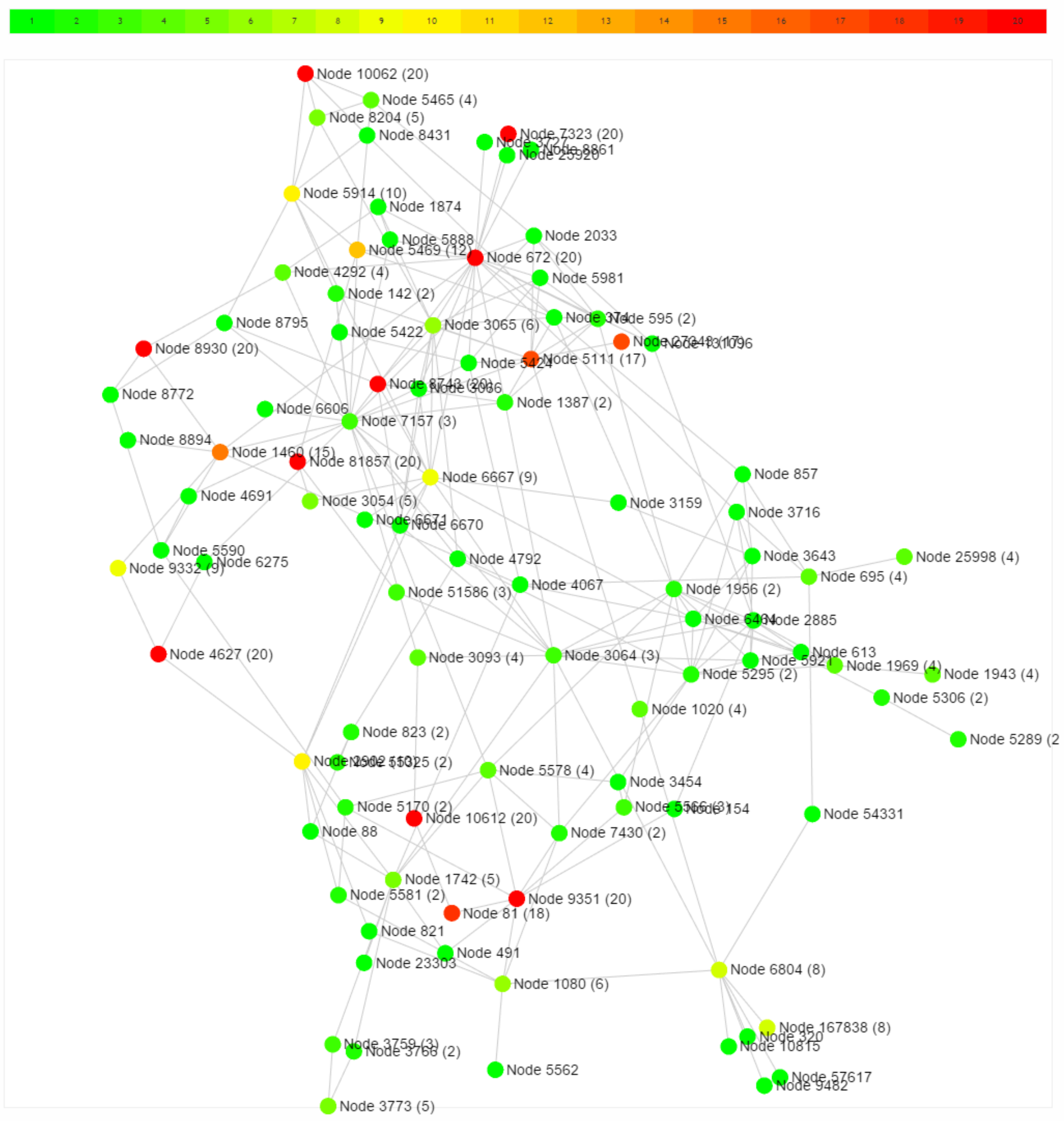


Figure 36: Network union graph of all results for a given k and l value.

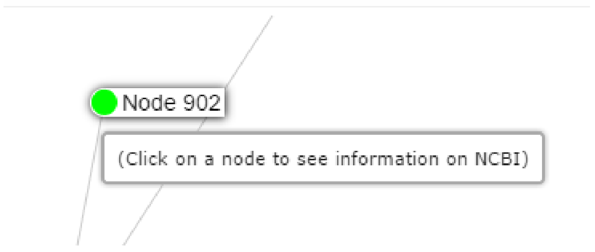


Figure 37: A node can be clicked, which will open the node’s page on the NCBI website.