8-2014

Pathway Pioneer: Heterogeneous Server Architecture for Scientific Visualization and Pathway Search in Metabolic Network Using Informed Search

Vipul Kantilal Oswal
Utah State University

PATHWAY PIONEER: HETEROGENEOUS SERVER ARCHITECTURE FOR SCIENTIFIC VISUALIZATION AND PATHWAY SEARCH IN METABOLIC NETWORK USING INFORMED SEARCH

by

Vipul Kantilal Oswal

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

UTAH STATE UNIVERSITY
Logan, Utah

2014

ABSTRACT

Pathway Pioneer: Heterogeneous Server Architecture for Scientific Visualization and Pathway Search in Metabolic Network Using Informed Search

by

Vipul Kantilal Oswal, Master of Science

Utah State University, 2014

Major Professor: Dr. Nicholas Flann
Department: Computer Science

There is a huge demand for analysis and visualization of the biological models. PathwayPioneer is a web-based tool to analyze and visually represent complex biological models. PathwayPioneer generates the initial layout of the model and allows users to customize it. It is developed using .net technologies (C#) and hosted on the Internet Information Service (IIS) server. At back-end it interacts with python-based COBRApy library for biological calculations like Flux Balance Analysis (FBA). We have developed a parallel processing architecture to accommodate processing of large models and enable message-based communication between the .net webserver and python engine. We compared the performance of our online system by loading a website with multiple concurrent dummy users and performed different time intensive operations in parallel.

Given two metabolites of interest, millions of pathways can be found between them even in a small metabolic network. Depth First Search or Breadth First search algorithm retrieves all the possible pathways, thereby requiring huge computational time and resources. In Pathway Search using Informed Method, we have implemented, compared, and analyzed three different informed search techniques (Selected Subsystem, Selected Compartment, and Dynamic Search) and traditional exhaustive search technique. We found that the Dynamic approach performs exceedingly well with respect to time and total number of pathways searches. During our implementation we developed a SBML parser which outperforms the commercial libSBML parser in C#.

(59 pages)

PUBLIC ABSTRACT

Pathway Pioneer: Heterogeneous Server Architecture for Scientific Visualization and Pathway Search in Metabolic Network Using Informed Search

by

Vipul Oswal

PathwayPioneer is a website to visualize the biological models. The complex metabolic models are hard to perceive when read through text format. It is even more difficult to understand the flow of network and different subsystem or pathways. PathwayPioneer organizes different subsystems into simple graphical representation. We simplify the complex network based on subsystem and compartment. Users can do computation (flux balance analysis) through the user interface and perceive the changed model visually. Our system is one of its kind to handle large model files with quick response time.

Biologists are often interested in knowing different alternative pathways targeting a particular metabolite. In the Pathway Search using Informed Search project we have developed four different search techniques which consider different properties of the metabolic network and eliminate redundant pathways. We have developed our own SBML parser which parses the SBML file much faster than a commercial libSBML tool. After comparative analysis of the search algorithms we found that the Dynamic Search algorithm performs faster than the other algorithms for finding alternative pathways.

CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION TO ARCHITECTURE OF PATHWAYPIONEER

PathwayPioneer is a web application which offers combination of Visualization and Computational analysis of biological models to a registered user. Models are represented using Scalable Vector Graphics (SVG) visual format and tabular data. The computation/-operations on the other had are performed on the model using flux balance analysis method. In addition, PathwayPioneer allows the visual layout of model to be modified as per the requirement of the user.

Users can upload their own biological model(s) in SBML or standard excel format on PathwayPioneer. The model is parsed, processed and converted into SVG format. The SVG depicts a layout of the model in the form of a visual hyper-graph. The layout is represented using a Map File which contains the co-ordinates of each graphical element of the SVG. Our tool can generate this Map File internally or can use one of the user-selected pre-defined Map File. Once converted into SVG, the model is ready for visualization and analysis. The tool facilitates the user to modify the current layout and customize it as per his needs. The modifications can be saved for future use. Other than providing the visual layout, the tool also summarizes the key attributes in the model like, subsystems, reactions in each subsystem, objective reaction, growth rate, total reactions and metabolites.

Users need to create subversion of the original model. Firstly this insures that integrity, format and visual representation of the original model is maintained. Secondly, it facilitates the users to create multiple subversion of the same model and work on each subversion independently.

PathwayPioneer allows users to download an original model or any of its subversion in SBML format. The downloaded SBML file contains all the modifications made to the original model. In addition it also allows to delete a subversion or entire model along with all of its subversion from the server.

As PathwayPioneer supports visualization and computation together, computation in the model is available through various commands. The computation can be directly done on the visual format of the model or can be done through the navigation menu provided. User can right click on the metabolite or reaction to use these functionalities. On a metabolites, user can edit its properties and navigate through all the subsystems in which the metabolite is present. On a reactions, user can change its flux, knockout the reaction, set that reaction as an objective-reaction or view its equation. Throughout the system user can also view the description of metabolites or reactions in KEGG and NCBI database. The flux-bounds of reaction can be modified individually or in a batch either on the hyper-graph visualization or inside the navigation menu.

PathwayPioneer offers a flexibility to undo these operations individually or in batch. The history of all such modifications is aggregated under Modification History tab.

Users can interactively study the model by zooming in and out of a particular subsystem or sub-section of the hyper-graph on the web GUI. One such snapshot of SVG layout produced and focused on a particular section of the hypergraph is shown in Figure 1.1. Reactions in which a metabolites of interest participates can be quickly searched in navigation menu and brought into focus. This search functionality reduces tedious manual search of a metabolite in large models.

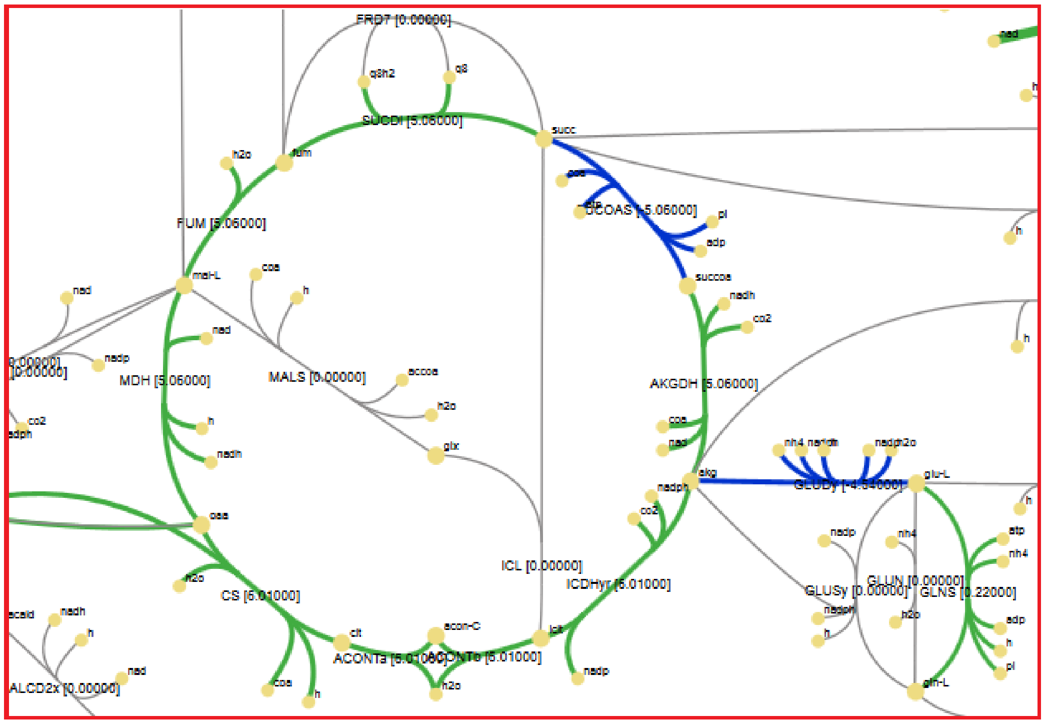


Figure 1.1: Layout section of a model EColiTextbook, after uploaded in PathwayPioneer

CHAPTER 2
RELATED WORK

Our current System runs on IIS 7.0 server on Windows Operating System. The server-side script is written in C#.net 4.0 using ASP.net Model-View-Controller (MVC) 3 web architecture.

2.1 Python and .net Server Communication

We use COBRAPy tool for modelling and flux balance analysis. COBRAPy [1] is a tool developed in python 2.7. Hence we need an efficient way for our .net webserver to make multiple parallel request to python engine. COBRAPy requires different prerequisites such as Numpy, Scipy, libSBML, GLPK etc.

Communicating with Python Engine from C#.net web Server could be done using various alternatives services libraries in the market. Some of the libraries are IronPython [2] and Python for .net [3]; which enables request invocation on python libraries through .net languages (C#, VB.net).

IronPython is an implementation of python programming language targeted for the .net framework. It does support python functionality however the prerequisite packages required for COBRAPy like Scipy are not supported in IronPython. Although Enthought [4] has come up with solution for Scipy in IronPython, it misses some modules such as Sparse matrix which are essential for Cobrapy. It should be noted that when IronPython was launched, it used its own interpreter libraries implemented in C# [5]. It was observed that any library module written in C may not work in IronPython.

Ironclad was introduced to allow IronPython users to use any compiled CPython extensions. However the library was targeted at IronPython2.6 and CPython 2.6 and is incompatible with COBRAPy which runs on python version 2.7. Moreover the development and maintenance of Ironclad is frozen since Jul 2010 [6].

Python for .net is an application scripting tool for integration of CPython with .net [7]. Therefore it is possible to use pythons native interpreter and CLR services for C-based extensions through .net programming languages (Managed C++, C#, VB.net, Jscript). However it is mainly focused on writing python script through C#. Although it does allow embedding python in .NET, the functionality is very limited to initializing engine, importing module etc. and cannot be extended for the specific use of PathwayPioneer such as handling standard input output, initializing multiple processes etc.

Hence, there was a need to develop a new stub to enable communication between .net and python servers which supports parallel request invocation from .net webserver to python engine, while allowing python to use its native interpreter.

.NET libraries provide inter-process communication through CLR. A new process of python can be instantiated through many programming languages in .NET. For PathwayPioneer we use C#. Therefore our interest lies in understanding the communication between C# and Python. System.Diagnostics is an assembly (library) that extends Process class and can be used to initiate different processes on the windows platform. The Process class API is also flexible to allow calling system operations such as Killing a process or acquiring process handle. Using Process class would be a good approach to bridge between C# and Python process. Process call when instantiate a python process, the python engine is launched by Operating System. Therefore we do not see the issues of CPython libraries and does allow separate engines to process their code & data separately. The communication between C# and Python happens through standard input output API of windows Operating system. As we can communicate between the processes, it makes system faster than the previous system where the output were written into files and read back by C# process when it is available.

2.2 Visualization Tools

There are plethora of applications available for visualization [8] of SBML and analysis [9] of biological model. However there are very few softwares which provides combination of analysis and visualization together.

Arcadia [10], a biological model viewer is a desktop application, therefore does not provide multiple simultaneous user access. Further it uses desktop resources which varies from users to users, thereby affecting systems performance. The focus of Arcadia is primarily on providing different perspective [11] on the same data. It uses Model/View/Controller pattern which makes it light weight. Users of this tool are confined to their system and cannot use it on any other system.

FAME [11] - the Flux Analysis and Modeling Environment is another web based tool which is used for modelling and flux balance analysis of stoichiometric models. User has to select a particular pathway or entire system to visualize. For every new change in the model, user has to upload a new model and rerun the FBA method. User cannot change the flux values dynamically and visualize the changes in the model. In addition, the layout generated for entire model is very complex and users cannot provide their customized layout.

AVIS [12] is another web-based cell signaling network viewer. It generates a layout of user model in the form of SVG. However similar to other tools it also generates a complex layout which is hard to perceive by users. Larger the model is, complex will be the visualization network.

CHAPTER 3
ARCHITECTURE OF PATHWAYPIONEER

3.1 Introduction

Current system runs on windows webserver IIS 7.0. The server side programming is written in C#.net 4.0 ASP .net. The website is developed in MVC (Model/View/Controller) architectural pattern. For biological calculations like FBA, we experimented using libraries on two different platforms:

1. Matlab engine using COBRA library
2. Python engine using COBRApy library

For using Matlab based library, a new instance of Matlab needs to be created for every new user. Creating a Matlab instance, initializing COBRA toolbox and loading user workspace are highly time expensive operations and uses more system resources when compared with using COBRApy library in python for carrying out the same tasks. Compared to Matlab, Python does not require COBRApy library to be instantiated. Adding the namespace of COBRA in Python is an instant operation. In memory model architectural pattern used for Python, eliminates delay caused by reloading model for each user request.

All model related activities (FBA) and web server management are completely separated. Since these two are separate processes or servers, communication mechanism is an important aspect for the system. Communication could be done through system IO operations like file reading and writing where each server writes its response in a file which is being watched by other server. As soon as file is updated the watcher server initiates a call to read a file. Other possibility of communication would be through inter-process communication using message passing paradigm. Communication through IO operation such as

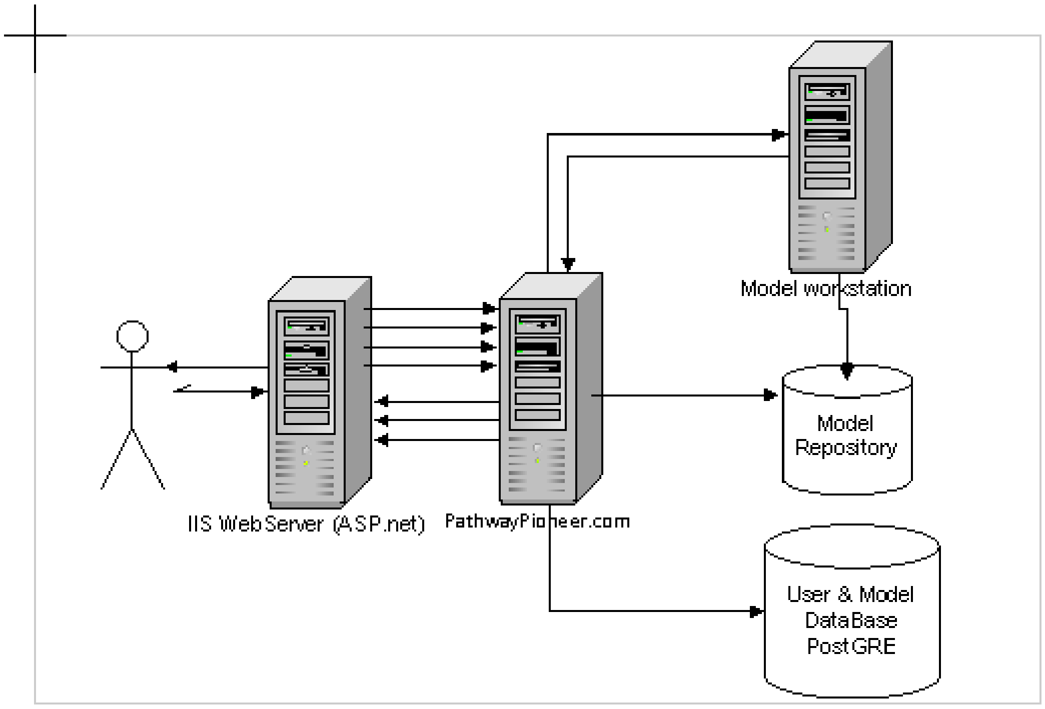


Figure 3.1: Component diagram of high level communication in PathwayPioneer

file read/write adds additional time latency to read and write by each process and involves system buffer operations. Therefore inter-process message passing method suits the needs, where the request and response is done directly through message passing channel. Although both the approaches require involvement of operating system, however time latency can be reduced by removing additional file read/write overheads.

Major architectural components in this project are

1. Web Server
2. Python Engine (Model Management)
3. User and Model Database
4. Model Repository

Figure 3.1 represents a high level system diagram where communication between different components is showed.

3.1.1 WebServer

Webserver uses ASP.net technology. IIS webserver handles every user request using its webserver lifecycle management component. It manages session for each user and passes the user request to the application. It handles multiple requests simultaneously and passes those in parallel.

Once the request is routed to the application, application parses the request and calls the appropriate handler to handle the request. The user management component authenticates the user and its request. If the user and the requests are valid, the application continues its normal functioning. It makes sure that user cannot perform any invalid operation on other users models.

For COBRAPy based requests, application forward those requests to the model workstation and waits for a response. User can upload a new model or can update the existing model. A new model can be uploaded with an existing or no map file. The application allows flexibility in terms of applying existing map file or use a model which is changed outside the application.

Once the model uploading request is received by the server, it stores the model in users repository and request is forwarded to the model workstation. Figure 3.2 shows different components present on the web server.

3.1.2 Python Engine

When python engine receives a request from web server, it looks up a user object in the global hashtable maintained on the python server. If the user object is present in the dictionary, it compares the model from current request and the model which is stored in the dictionary for current user and performs serialization/deserialization if required. Python server handles multiple requests simultaneously using multi-processing paradigm. Although python supports multi-threading, it uses global lock (GIL) on the python script when a particular thread enters a script, this behavior of Python language does not scale the application very well and adds time latency since other threads has to wait in a queue unless current thread finishes. This prevents from true parallelism. Multiprocessing allows

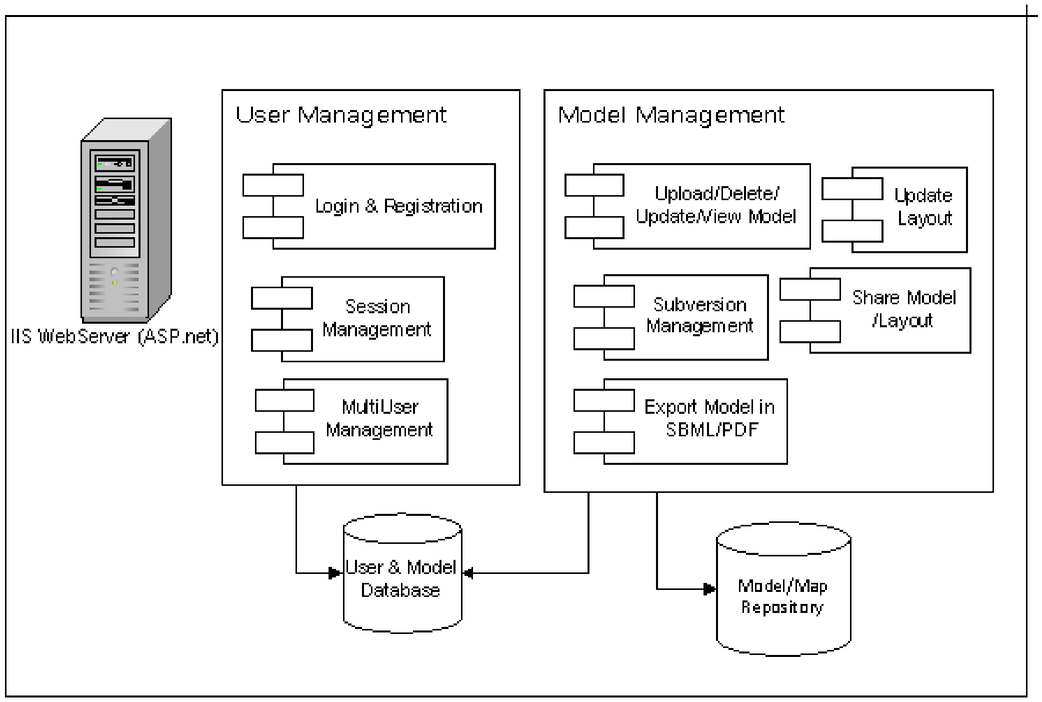


Figure 3.2: Component view diagram of web server and application

parallelism by using all the cores available on system hardware. In this architecture we maintain three important components. ProcessPool, WorkQueue, UserMap.

1. A ProcessPool is a list of spawned python processes to serve users request. The total number of processes is a dynamic number and varies with the total number of CPUs/Cores.

2. A WorkQueue is a global shared queue on python server. All of the requests from the webserver are enqueued here. These work items are consumed by free processes from ProcessPool. The work item in WorkQueue could be to upload a new model or to do flux balance analysis on the current model.

3. UserMap is a hashtable of currently active users objects. It contains user related attributes such as user ID, currently accessed model etc. The UserMap can be shared by multiple processes simultaneously, however only one process can work on users model to avoid model reloading conflicts. Serialization and Deserialization are done only when a new work item requires a model which is different than a model requested

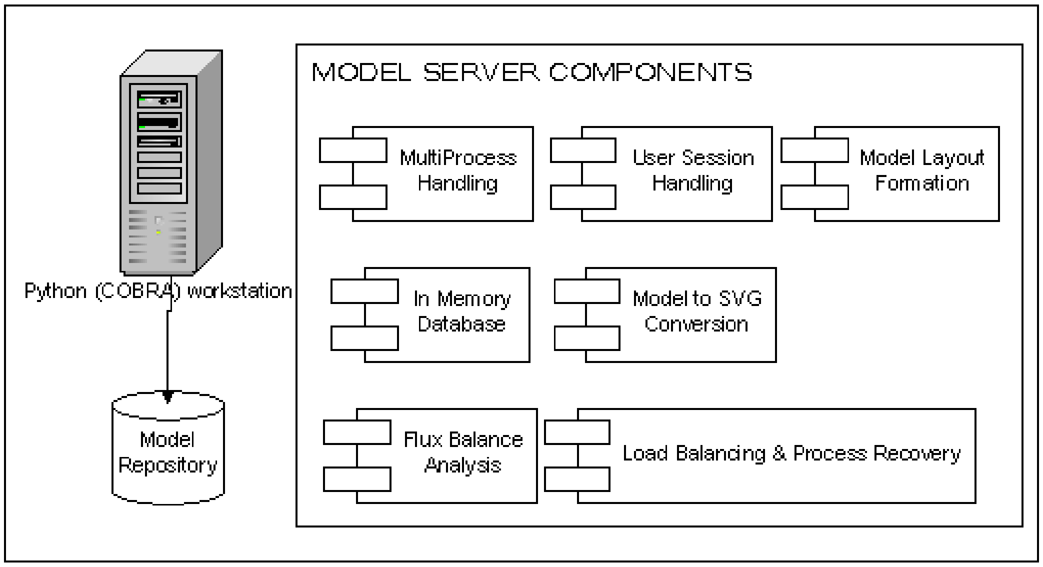


Figure 3.3: Component diagram of Python server (Model Workstation). It handles several operations related to model handling and multiprocessing

in current work item. Limiting serialization and deserialization operations saves model loading time or writing model back to a file.

The scenarios in which communication between webserver and python engine is required are:

1. Upload a new user model
2. All Flux Balance Related Operations
    (a) Knockout Reaction
    (b) Edit Fluxes
    (c) Set Objective-Reaction
    (d) Revert Changes
3. Edit existing layout
4. Download user model

Different model related operations are described in figure 3.3.

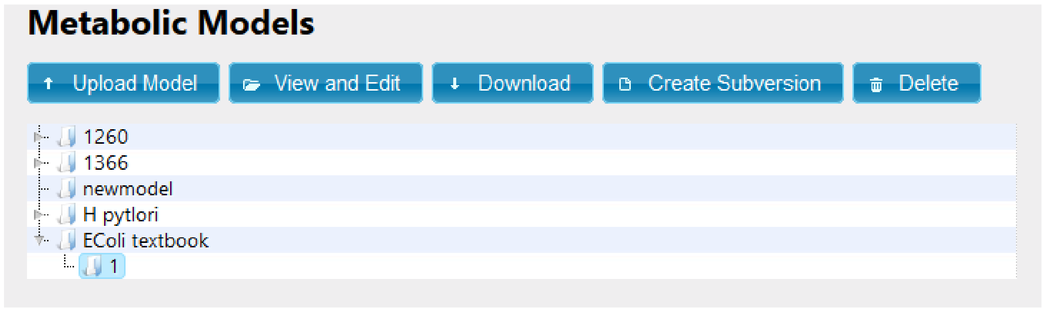


Figure 3.4: User repository in PathwayPioneer, the tree represents the lists of models used by user. Different operations can be performed after selecting a model or its subversion

3.1.3 User and Model Database

Each user has its own repository. The repository contains all of the models they are using including their subversions. Please see Figure 3.4. A typical model repository contains the SBML/EXL model, map file (Layout file), SVG of a model. In addition there are also repositories for global models and map files. Users can publish their map files and share it with other users. The global model repository contains the popular models used by users. These models are ready to be used by users.

The postGRE database containing user information and model information and their association.

3.2 Coding Description

PathwayPioneer.com is divided into three tiers. Client facing user interface, WebServer (IIS and .net) and backend Python Engine. In addition we also use PostgreSQL for data storage.

Client facing user interface consists of HTML pages generated by IIS server, strong binding of pure JavaScript and JavaScript libraries UI. It renders SVG file generated for user if user uses repository services offered on the website. To make the best of user experience by providing slick and flexible UI components, different JavaScript libraries are used. JQuery is one of the major and heavily used for binding different context menus, zooming the SVG components, binding SVG components to the menus and tooltip. JStree is used to populate user repository and display those in the tree structure.

The extensive use of these libraries allows to make majority of the communication asynchronously with the server. Therefore making user actively engaged with the UI. The user interface is a heavy weight process, and includes complex JavaScript code to render SVG and computation of real time attributes for SVG. To make UI more flexible, model data is represented in the graphical and tabular format.

Heavy SVG files could contain more than two million lines; therefore loading them on browser becomes a critical issue. Browser process may crash due to unavailability of the memory or could take long time to load the SVG. Hence, only one subsystem from the SVG is processed at a time and that subsystem becomes a focus of user. Other subsytems are loaded in real-time based on user demand.

JQuery context menu is bound with each reaction and metabolites. Therefore right clicking on the reaction or metabolites of SVG provides various possible operations. These operation are JavaScript functions, which in turn sends requests to the webserver. All the FBA functions are synchronous in nature due to the fact that, the results returned by server not only makes changes to the focused entity but it can also make changes to the entire model, therefore SVG needs to be regenerated. However uploading a model does not require a synchronous call and therefore a user can upload multiple models simultaneously.

3.3 Web Server (C#, ASP.net, MVC)

The webserver architecture is based on MVC [13] pattern (Model/View/Controller). MVC is a three-tier architectural pattern; however differs from traditional three-tier architecture pattern [14]. The MVC pattern has three components. 1) Models, which manages and implements logic of Data tier. It stores and maintains users and datas object state. 2) Views, target at implementation of user interface (UI). It displays the data in the Model state in different formats. 3) Controllers are central unit that manage user inputs and query models to get the next relevant state. It renders appropriate view once the model returns or it computes new UI state. In MVC, views are pure UI components.

MVCs loosely coupled components allow to make the software development modular, extendible and highly scalable. Because of loose coupling, it is easier to make changes independently in all the three layers. MVC applications in ASP.net provides a routing mechanism of URL. Requested URL received at the web server is routed by IIS to the web application. At web application, global.asax is an application level configuration file, parses the URL and checks if the URL is registered. If none of the registered URLs matches with the requested URL, the error view is returned. This door step URL handling makes PathwayPioneer really robust against malicious requests sent by anonymous users. If the URL matches with any of the registered URL, the request is routed to the appropriate controller for suitable action.

In PathwayPioneer, we use the model data in different formats and store them in separate objects. Hence, any subsequent changes in model data does not affect other layers and thus changes can be made independently. We have divided views and controllers in five sections namely Home, About, Contact, Repository and SVG. Out of these five sections, Repository and SVG controllers are major components with respect to total number of functionalities offered by them.

Our implementation has three primary controller classes. Account Controller handles all user actions such as register a new user, login, logout and other user authentication tasks. Repository Controller manages tasks associated with users repository such as upload new model, publish model or map file, reload existing model, update map file of existing model, download model, create a new subversion of model, delete model. These tasks, except uploading a model, run asynchronously. The results are returned to client-side Javascript libraries in light-weight JSON format. SVG controller manages all the operations carried out on SVG model. These operations include Flux Balance Analysis operations (edit flux, knockout and set objective), editing and updating model layout, reverting any of the SVG changes. For scalability, we have extended all the FBA functions which can execute multiple FBA requests in one transaction. Therefore one can edit multiple reaction fluxes in one operation or can revert multiple previously executed operations.

The controller handles the input and processes the request using model data stored in users session state. However it does not parse or store any of the user data. The model data is stored in .xml file in sbml format. PostGRE admin database service is used to store the user account data and all the relational data. This data includes users repository, their models, the operations done on the models, their subversions etc. We have normalized the databases and created separate tables for models, maps, user accounts, operations etc. Further, we have created SQL stored procedures to publish the models and maps by users. The stored procedures isolates the query details from the implementation code by exposing only meta-data like the name and the parameters required to execute it.

Python Service is an interface to invoke parallel requests from webserver to Python engine and receive corresponding results. These request calls are required to use features like FBA in COBRApy library. We implemented this interface using standard windows API. Following, singleton design pattern, a single instance of python process is created for the entire lifetime of the application. Python engine is initialized with the configuration like, importing system library, setting the windows system PATH variable to destination python script directory, etc. Using windows standard input/output API, request-command messages are sent by webserver to python engine in the following format:

OperationName,ModelPath,UserID,ParentID,VersionID,Parameters

Where,

OperationName $\rightarrow$ Action to be taken on model

ModelPath $\rightarrow$ Directory Path where model,map file is present

UserID $\rightarrow$ Unique ID associated with the User who has requested this operation

ParentID, VersionID $\rightarrow$ Candidate combination representing a model

Parameters $\rightarrow$ Differs for every operation.For edit flux operation on the reaction TKT2, the parameters would be TKT2:¡new lower bound¿:¡new upper bound¿. If the operation is knockout the values of lower bound and upper bound is set to zero. To set a reaction as an objective-reaction, the parameter is the id of that reaction. Multiple parameters can be passed in CSV format.

Example: EditFlux,C:/20/245/234/,20,245,234,TKT2:0:0

The format of the response messages from python engine to webserver is:

UserID:::SuccessFlag

Where,

UserID → unique userid of user who has sent a request

SuccessFlag → return 1 if the operation was successful and -1 if it is unsuccessful.

Example: 20:::1 or 20:::-1

Mutex locking mechanism ensures synchronization of commands and prevent multiple parallel commands from interleaving with each other on the output buffer.

Python Service provides output queue shared by all users to buffer responses messages sent by Python engine to web server. It allows synchronous and asynchronous request-commands to be sent to the python engine. The synchronous commands make a blocking call for that user and awaits for a response in the output queue or a timeout period to expire. The timeout period is managed by expiration protocol timer to avoid infinite loop exceptions. Operations like Edit Flux, Knockout, Set Objective, Revert changes, etc. are carried out synchronously, as these operations modify the flux values of the reactions and hence requires the SVG to be updated. For asynchronous requests, the webserver does not wait for the response and unblocks the users UI. In the background, the progress of the asynchronous operation is monitored. Once the operation is completed, the computed response is sent to the user through AJAX. The Upload model operation is carried out asynchronously. Upload model is a time consuming operation and may take few seconds to several minutes to upload a model and make it available for user. The time required to upload a model is based on the size of the model. A simple E Coli Textbook model having 70 metabolites and 92 reactions takes up to a few seconds to make the SVG of the model available. However models such as E Coli iAF1260 or E Coli 1366 usually take up to a minute or two to generate the SVG. The computational time required increases linearly with the model size. Since the model uploading operation is asynchronous, user can do other tasks on existing models simultaneously while model is still getting ready in the background. Users are also notified by email once the model is ready to be used.

Two events in python service, ErrorDataReceived and OutputDataReceived, notify the webserver once the python engine responds back with an error or output to an invoked request. The event handling is a Push notification service. The output messages are pushed from python engine to web server. Therefore webserver saves CPU monitoring time while the python engine is processing its requests.

Currently, it is not possible for python engine to respond with appropriate error text to webserver to allow the controller to corrective actions. Nevertheless, any exception in python are logged for troubleshooting purpose.

3.4 Python Engine Scripting

Python uses Global Interpreter Lock (GIL) which prevents true parallelism. GIL prevents the benefits of using multi-core system and only virtual parallelism can be achieved. It is similar to running a multithreaded application on a single core machine. Therefore to achieve true parallelism we use multiprocessing paradigm offered by python. Multiprocessing spawns new python processes/sub-processes. This would not only allows us to make best use of hardware with multiple cores but it also execute the code independently with its own set of local variables. Inter-process communications are more computationally intensive in comparison to inter-thread communication, which is a major drawback of shifting to this paradigm. However, with our architecture, the processes do not need to communicate with each other as each process works on independent request-commands from webserver and free themselves when finished processing.

Python Engine implements its own process pool model to make best use of multiprocessing. The multiprocessing pool model consists of three major components - Global User States, Process Pool and Work Queue. The detailed architecture is as shown in Figure 3.5.

Global User State: This is a customized global hashtable data structure maintained on python engine where the key is the unique userID and the value contains the information of the model. Thus, upon receiving request from any user, the corresponding model information can be retrieved in constant time. Before processing the new request, the model for which the request is being made is verified against the model retrieved from the hashtable. If the request is for a different model, the old model is first serialized at relevant repository

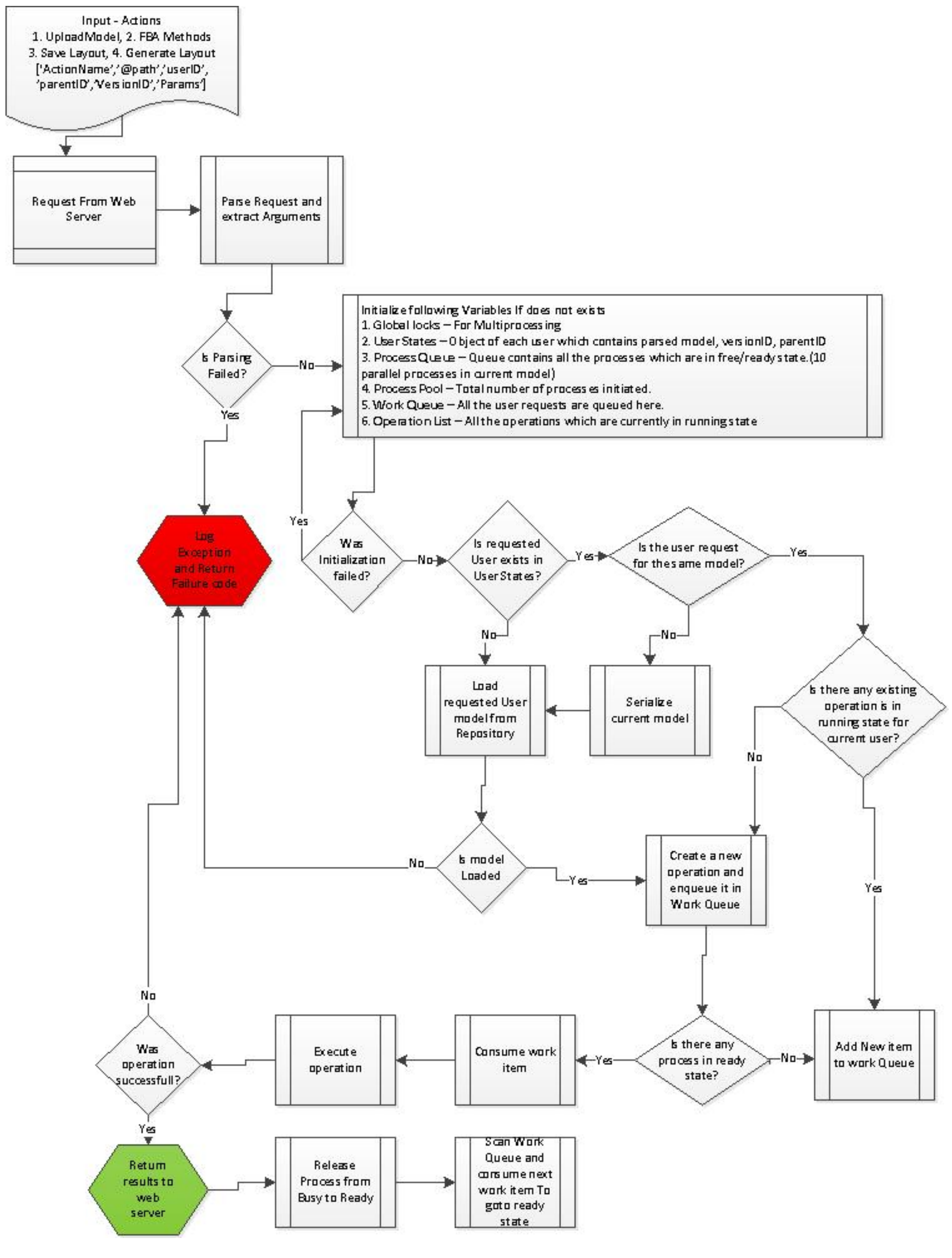

Input - Actions
1. UploadModel, 2. FBA Methods
3. Save Layout, 4. Generate Layout
['ActionName','@path','userID',
'parentID','VersionID','Params']
Request From Web Server
Parse Request and extract Arguments
Is Parsing Failed?
No
Yes
Initialize following Variables If does not exists
1. Global locks – For Multiprocessing
2. User States – Object of each user which contains parsed model, versionID, parentID
3. Process Queue – Queue contains all the processes which are in free/ready state.(10 parallel processes in current model)
4. Process Pool – Total number of processes initiated.
5. Work Queue – All the user requests are queued here.
6. Operation List – All the operations which are currently in running state
Log Exception and Return Failure code
Yes
Was Initialization failed?
No
Is requested User exists in User States?
Yes
Is the user request for the same model?
Yes
No
No
Load requested User model from Repository
Serialize current model
Is there any existing operation is in running state for current user?
No
Yes
No
Is model Loaded
Yes
Create a new operation and enqueue it in Work Queue
No
Was operation successfull?
Execute operation
Consume work item
Yes
Is there any process in ready state?
No
Add New item to work Queue
Yes
Return results to web server
Release Process from Busy to Ready
Scan Work Queue and consume next work item To goto ready state

Figure 3.5: Detailed Data Flow (DF) diagram on Python Engine.

space and the model for which the operation is being requested is deserialized into the hashtable for processing. It is observed that most of the subsequent requests from a user are for the same model. Thus by deserializing un-used models avoids storing all models in the memory giving a significant ease on the memory resources. In addition, through serialization and deserialization object state is automatically maintained without the need of additional parsing logic.

Work Queue: It is queue data structure that en-queues the webservers request-commands. Request-commands are represented as an object of UserOperation class storing all the required parameters from the command. Mutex locking ensures mutually exclusive access to this queue. Request-command is received in the form of system arguments (sys.argv) and it is essential to prevent replacement of these values in by the new request through locking. After parsing the parameters from sys.argv and converting them into an object of UserOperation class, the object is en-queued in the WorkQueue and the mutex is released.

Process Pool: This is a list of processes spawned by original python process. The total number of processes spawned is directly proportional to the number of cores available on the hardware resources (CPU) of the server. Each process is initialized to the ready state. Items from work queue are assigned to the process by de-queuing it from process pool. The process once started, sets itself in busy state. It independently processes the work-item and returns the result. It then checks for any unprocessed work item in the WorkQueue and starts processing it. If the WorkQueue is empty, it sets itself back to ready state.

Our multiprocessing module conceptually works in the similar manner as that of Producer Consumer problem. A typical request-command is processed as shown in Figure 3.6.

3.5 Testing PathwayPioneer

Testing a web application becomes crucial, when it handles concurrent users with large amount of user data. PathwayPioneer mainly uses model data and performs different op-

```
100 The request-command arrives to python engine in the form of sys.argv.
101 Global System resources are initialized if not available.
102 New Work operation is generated and enqueued to work queue.
103 Process P is fetched from process pool if available.
104 Work item is fetched from WorkQueue.
105 P is assigned with the work item.
106 P starts and sets itself to busy state.
107 P processes the request and returns the result/error.
108 P checks if work-item is present in WorkQueue. If yes, goes to step 104.
109 P sets itself to ready state and enters ProcessPool.
```

Figure 3.6: Request-commands processing

erations on it. These large models reside in main memory when accessed by users. Every time user performs an operation on the model, the SVG is regenerated and rendered on the browser. Therefore response time increases proportionally with size of the model. So to measure the quality performance of PathwayPioneer in terms of response time, availability and scalability, we perform concurrency test on PathwayPioneer.

Our goals of testing PathwayPioneer is to check how the performance of the system is affected by various loads on the server by testing the scalability of PathwayPioneer for multiple simultaneous users and by analyzing the performance of the system when multiple users simultaneously use similar functionality and when they use different random functionalities.

Heavy use of JavaScript code and libraries, dynamic production of different SVG elements, unavailability of unique ids to some SVG elements increases the challenges in automating the tests for the website. For eg. To select a model in UI through code is difficult as model element do not have unique ids and it is difficult to traverse and retrieve the exact element as new elements get added dynamically at run-time.

Web Application Toolkit In .Net (WATIN) 2.1 [15], 3rd party library available for .net web testing allows the testing script to be developed using C# programming language. WATIN supports the use of Firefox client for testing the web-application.

To perform automated tests of our tool, various testing toolkits available in the market were studied. Selenium [16], a browser automation toolkit for automation web UI testing, does not capture the dynamic models added into the repository tree and it does not select elements of SVG for FBA function. Other tools and libraries like WebBrowser toolkit of Visual Studio, not only faces the same problem as Selenium but also fails to test a website on Firefox. One of the major issue of simulating multiple users from same system is to maintain independent sessions for each users in different browser processes. This is not a default behavior of modern day browsers as session information is shared across the browser processes. To the best of our research and knowledge, there is no solution to create and maintain automated independent sessions of Firefox browser. The private session feature provided by Firefox does initiate new session windows, however it limits two sessions at a time per system.

Similar to Firefox, Internet Explorer (IE 10) does not create isolated sessions by default. However this behavior can be altered by one-time configuring registry settings of IE. If registry dWord SessionMerging is set to 0 than each window of IE will isolate its session state and cookies. Hence different users can authenticate and access our tool independently from separate windows of IE from the same system.

Certain controls like file input control requires exclusive focus to set its value. The operating system allows only one element to be in focus at a time. WATIN uses a browser class to set focus on the HTML element of interest and set the value to the element on the fly. Therefore in multithreading environment, concurrent multiple users on the same system need an exclusive access mechanism while setting the values of such controls.

Multi-threading in C# to handle multiple concurrent users fails as IE processes are opened by the same main process. As a result, IE does not isolate its session and cookie states for separate users. To overcome this problem, each users test process was assigned to a separate process by multiprocessing technique of C#. Each process of a user runs the

Upload Model Script

```
Load PathwayPioneer Web project
Login test[i] user, where i is different for each browser instance with
    different session.
Upload a model EColi Textbook simultaneously by each user.
Log the upload model time
```

Figure 3.7: Upload Model Test Script

end-to-end testing script independently and log the results.

For testing purpose we hosted our server on the system comprising of a Windows 7 64-bit machine having 16GB DDR3 main memory and Intel core i7 processor with 3.45 GHz clock cycle. We use default IIS process pool.

On client side we use Internet Explorer 10 with configured registry settings. To avoid additional overhead on server, each client machine maintains the logs of the users running on them. Each client machines comprised of Windows 7 64-bit, i7 processor with 8 cores and 8 GB main memory. We used four different workstations of same configuration. Our target is to check the load and functionality for 50 simultaneous users.

The primary testing script is as shown in Figure 3.7 and 3.8.

Above testing scripts are automated to check the load behavior of website during different operations. In first scenario multiple concurrent users simultaneously uploads model in their repository. The model used for uploading was E-Coli Textbook. Once the model was uploaded, the results are logged. Since the allocation of cores to processes is managed by operating system and due to limited number of cores on hosting server, the performance of the tool degrades with more number of users trying to upload a model in parallel. Uploaded model time also includes first time initialization of python processes and other python engine resources. Therefore uploading a model for even 6 users took higher time than expected. However when 50 users are uploading models concurrently, many of the processes finishes quickly since the process initialization step was executed during the

Markov Model Test Script

```
100 Run PathwayPioneer project
101 Select a random operation from a queue (Upload Model, Knockout, EditFlux)
102 Add a random delay to the operation
103 Execute the operation
104 Log the operation time and results
105 Repeat step 101−104 until 25 minutes lapse.
```

Figure 3.8: Test script to randomly select an operation on PathwayPioneer for random testing

initial upload model requests.

Secondly, the Markov testing model scenario covers random testing of PathwayPioneer. This is from the assumption that the real users would not do similar operations in parallel. They would do some random operations with random additional delay between two consecutive operations. Figure 3.9 shows state transition diagram in which the probability of state transition for users to switch from its current state to next state. Our probability matrix for state transition is as below. As per the observation on PathwayPioneer the probability of transition from different operations is represented as.

$$P = \begin{pmatrix} 0.02 & 0.4 & 0.58 \\ 0.09 & 0.3 & 0.61 \\ 0.05 & 0.55 & 0.4 \end{pmatrix}$$

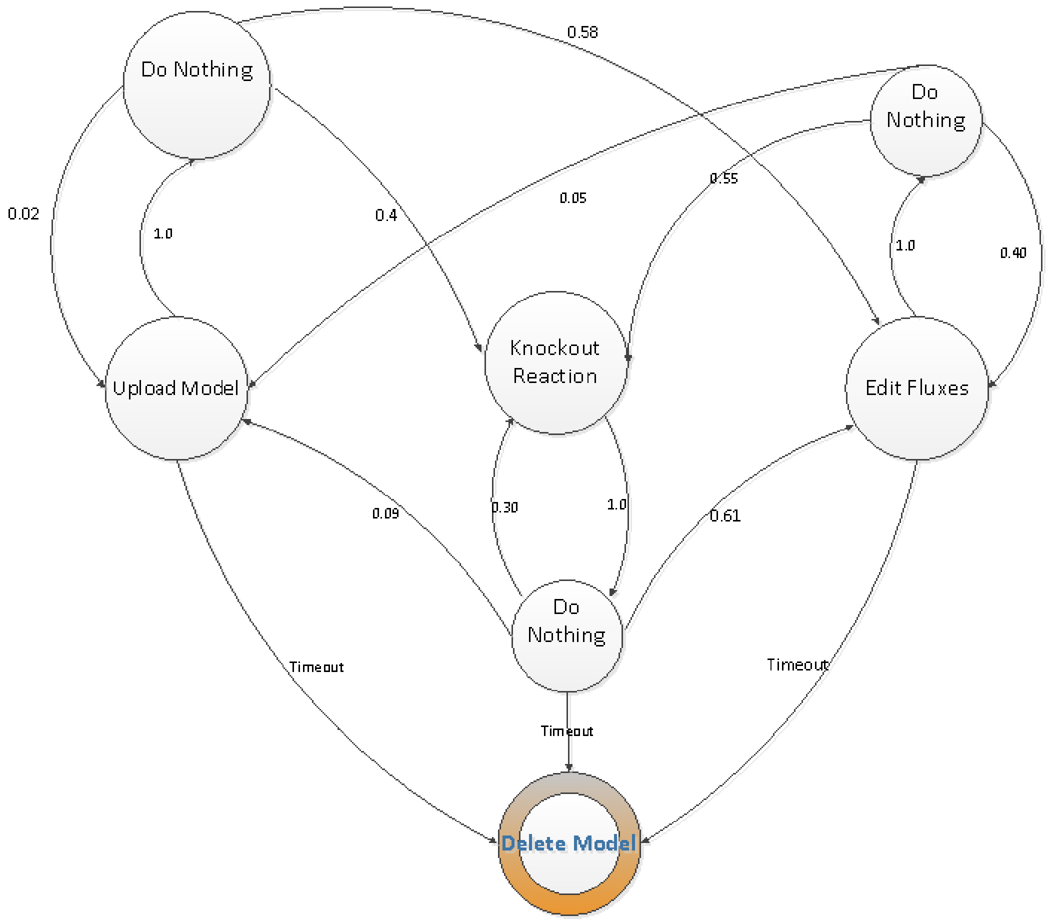


Figure 3.9: Markov state transition model, for testing the scalability by performing the random operations on PathwayPioneer.

CHAPTER 4
RESULTS

4.1 Upload Model Test

The upload model operation performed by 50 concurrent users shows that, as the number concurrent user grows, the upload time increases as shown in Figure 4.1. This is from the fact that, while python processes serves the in-hand request, the other requests are en-queued and waits for processes to get freed.

4.2 Markov Model Test

The Markov testing model however shows that when the user behavior on PathwayPioneer is simulated and state is changed with a random probability, then the website shows its nearest best performance for each operation. The average time for each tested operation in Markov probability model is shown in Figure 4.2, 4.3, and 4.4.

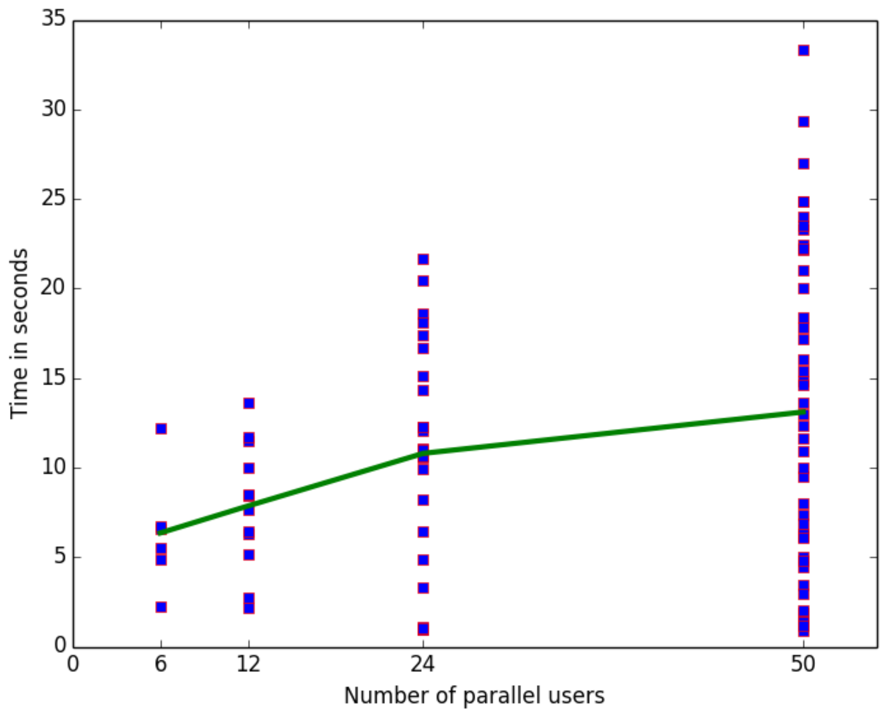


Figure 4.1: Parallel upload model testing. We test PathwayPioneer for concurrent/simulataneous model uploading task, we have used E.ColiTextbook model and run the website with multiple parallel users.

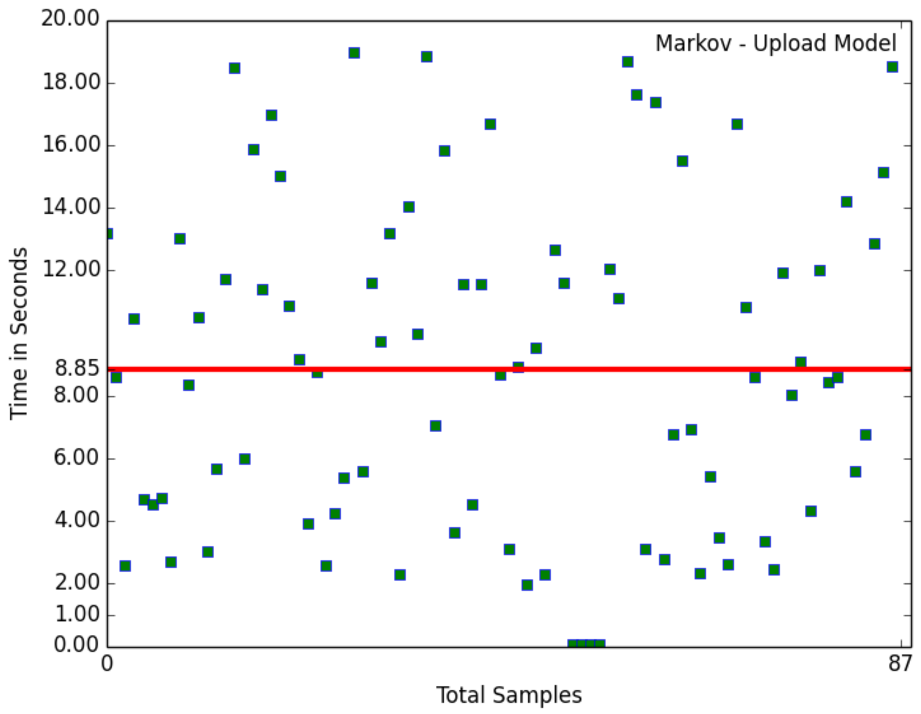


Figure 4.2: Average time observed for upload model operation in Markov testing model. Markov testing model is executed in parallel by different browser sessions. Note that in the beginning of every test sample, each dummy user uploads the model first and then perform random operations either on the model or upload a new model once again. Any subsequent edit-flux or knockout operation is performed on last uploaded model. The time taken by upload model operation here does not consider the very first upload Model action. But it considers all subsequent upload model operations happened for that user. We run this test for 50 parallel users and observed that in Markov testing model, the intermediate upload model operations perform much better than that of simultaneous upload model operations performed by users. The average time to upload a model in this scenario is 8.85 seconds which is much lesser than 14.56 seconds time observed while parallel model uploading test scenario. As compared to knockout and edit-flux operations, we get very less samples of upload model due to the fact that it has the lowest probability of transition from any other states including itself.

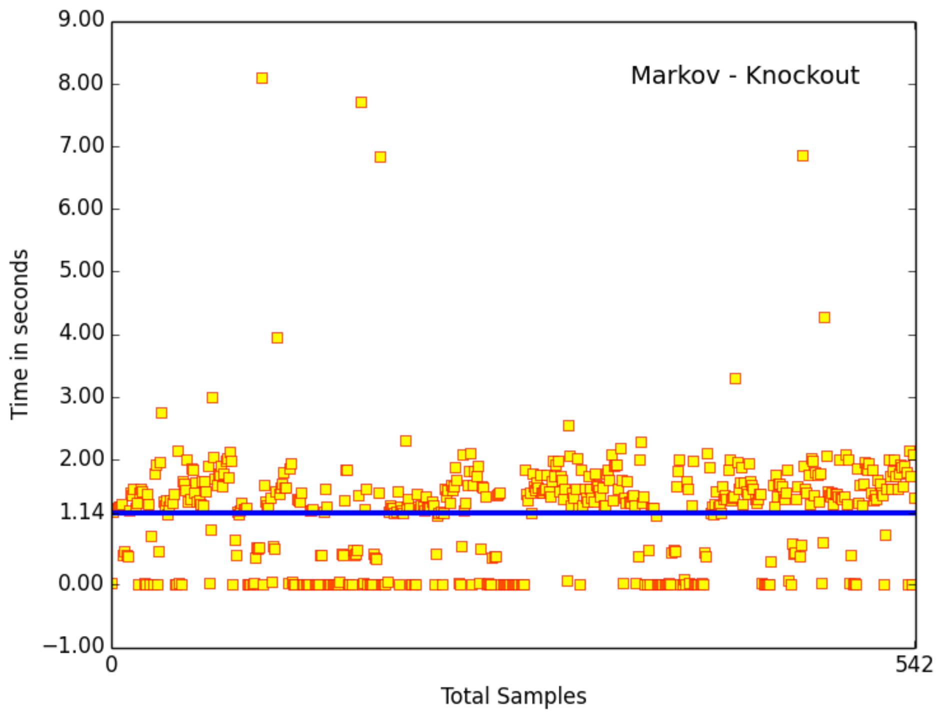


Figure 4.3: Each dot represent a knockout sample collected while testing PathwayPioneer using Markov modelling approach. We have collected 542 samples from 50 independent dummy users (via separate browser sessions) running on 4 independent workstations, continuosly for 20 minutes. The average knockout time observed was 1.14 seconds which is close to the single user manual testing sample. We do not add any additional delay (SVG file writing time overhead) to the total knockout time. The function returns as soon as the knockout operation is performed on the in memory model object. Therefore on an average website receives response back in 1.14 seconds after a knockout. In reality, browser may takes additional time to render the SVG. In above graph, we can see some of the samples which are outliers, note that the exceeded time can be improved by using higher configured server and adding more processes in parallel processing of python engine.

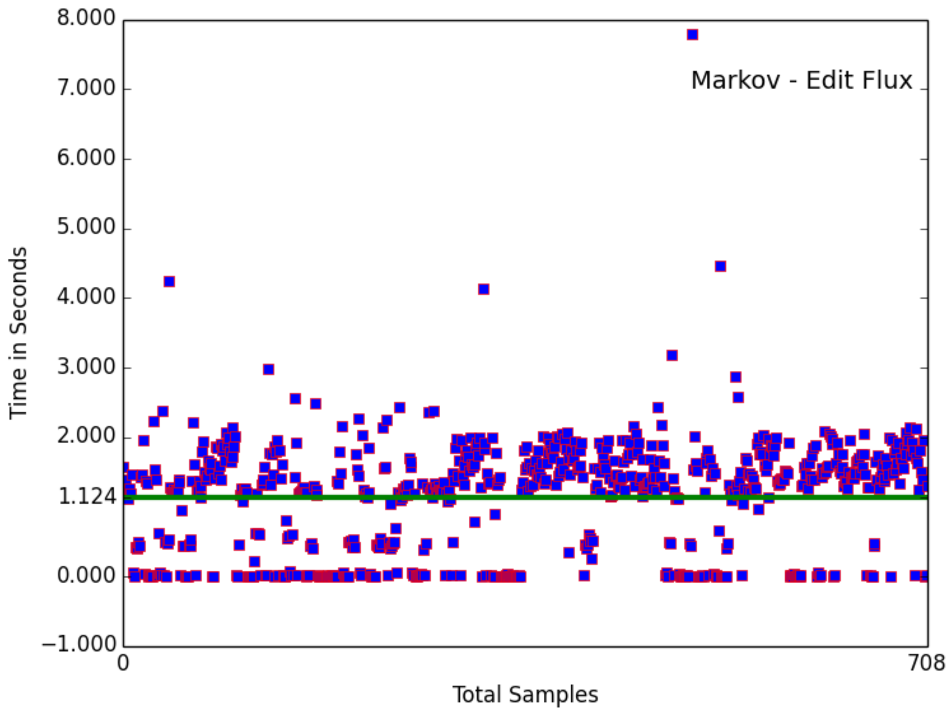


Figure 4.4: Edit flux is one of the FBA operation performed through COBRApy library. The graph shows performance of edit flux function in Markov texting approach. Although edit-flux and knockout are similar operations in terms of flow of the data, probability of continuously selecting the same operation reduces. Furthermore, edit-flux is one of the important operation frequently performed by users. Therefore we added the edit-flux operation to our test scenario and observed the performance against the time. The testing process logged over 700 samples of edit-flux operation. We see that the average time required for the edit-flux operation is 1.124 seconds. Similar to knockout, we do not add any additional time required for writing SVG file. Further all of the above sample operations are performed on the model E Coli Textbook.

CHAPTER 5
INTRODUCTION TO PATHWAY SEARCH

5.1 Introduction

A metabolism plays very crucial role in the life of an organism. Metabolism is a system through which several essential compounds/metabolites are produced such as amino acids, through chemical transformations/reactions. A continuous transformation of metabolites gives very essential information about their evolution in the system. Engineering this system plays an important role in understanding the role of bacterias and their evolution.

A metabolic network consists of set of reactions interconnected to each other through metabolites. A given reaction takes multiple metabolites as an input called as reactants and produces multiple metabolites as an output called as products. A series of such reactions can form a single pathway. Different pathways could share the same metabolite, therefore there may exist multiple pathways between two metabolites. A metabolic network can be represented using SBML structure. SBML is a markup language and represented using tags and relation between the tags (Similar to XML). SBML can be treated as a graph G (V, E) where all metabolites can be represented as nodes (V) and the reactions (E) between them is treated as edge. The direct conversion of SBML to graph produces a Hypergraph H (X, E), where X is a set of metabolites or set of nodes and reaction E is the hyperedge, which connects any number of nodes/metabolites may or may not include cofactors.

5.2 Terms used in the rest of the paper

SBML Systems Biology Markup Language, an XML format for storing biological computational models.

Metabolites represented by species in the SBML, more details on metabolites can be found at [17].

Reactions A chemical reaction takes one or more metabolites as an input and produces one or more metabolites as an output

Cofactors A type of metabolites takes part in reaction to assist the reaction or produced as a byproduct. Further details can be found at [18].

Pathway A series of biochemical reactions which transforms metabolites inside cell.

Hypergraph A graph in which an edge can connect one or more nodes.

CHAPTER 6

RELATED WORK

Many search-based algorithms have been researched by researchers to find pathways in metabolic network. PathMiner [19]: predicting metabolic pathways by heuristics search is a tool which uses A* algorithm to find the shortest pathway between two compounds (metabolites). It transforms a metabolite network into state-space where compounds are represented as a state and transformations are represented as transitions. The approach they have used is different in terms of the heuristic function used (based on Kegg [20] database), the data structure used and the transformation of the problem into state space.

Küffner et al. [21] in his paper showed that the performance of a search algorithm not only depends on the size of the network but also on the parameters intertwined with the structure. He showed that given a metabolic network, the total number of unrestricted paths connecting glucose and pyruvate is around 500,000 which is potentially a very large set. Therefore author computes valid pathways by pathway generation algorithm and use DMD (differential metabolic display) to deduce all of the valid pathways in a given network. However this method certainly requires additional time latency to generate an original graph and later differential graph. Kffner used Petri Nets in metabolic network for directed bipartite graph representation and applied branch and bound algorithm. However he found that very large set was retrieved (500,000 paths from glucose to pyruvate), their DMD (firing rule) reduced the paths to 80,000, this was further reduced to 170 pathways by further conditions. Because of such a large number of paths retrieved, the further research was identified and it was based on the shortest path search.

In shortest path research, the shortest path is defined as the smallest numbers of transformations required to convert given metabolite into a goal metabolite. Or shortest number of reactions involved in a pathway.

Zhou and Nakhleh [22] talks about the modeling of biological network in Properties of metabolic graphs: biological organization or representation artifacts. The paper talks about whether to represent the metabolic network as a standard graph, where every edge shares 2 nodes or represented as a hypergraph. The author found that the scaling is the major issue during clustering in a hyper graph.

Faust et al. [23] [24] in their paper Pathway discovery in metabolic network by subgraph extraction discusses about different approaches to extract all the relevant pathways from metabolic network. They used hybrid strategy, based on random-walk reduction of the graph with shortest-path algorithm.

McShan et al. [19] convert a metabolic network in a graph where they consider reactions taking only one metabolite as an input and one metabolite as an output, ignoring substrates. They apply A* search to find the pathways and claims it works better than DFS or BFS.

Croes et al. [25] [26] use depth first backtracking algorithm to find shortest pathways. They define a degree of each metabolite by computing total number of reactions in which metabolite participates either as a product or reactant. By choosing low degree nodes they find results are much improved.

Rahnuma: Hypergraph based Tool for Analysis and Comparison of Metabolic Pathways [27] is another tool which analyzes and predicts the pathways in the Hypergraph using structural analysis.

Many of the existing algorithms use a conditional search based on the k shortest paths. Biocyc [28] is one such tool and can be used to find k (maximum 100) best routes between two compounds/metabolites. It provides an interface to provide start and goal metabolites. However it uses a limited search and requires few conditions to be defined such as reaction cost. Similarly there are many of such applications which work on limited set of results. The pathways search in these applications is specifically based on the database they maintain and does not work on SBML model which can be uploaded.

In rest of the paper we will discuss about the graph representation of the metabolic network, various properties of this network and how those are useful, algorithm used to search pathways between metabolites, results through different search approaches and conclusion.

CHAPTER 7
METHOD

7.1 Modelling Metabolic Network

SBML [29] is de facto standard for representing biological computational models. In this project the input of a biological model is taken in the form of SBML file. Figure 7.1 shows a block diagram of a SBML Parser. After parsing the SBML file, a collection is created from the list of metabolites and their associated properties like compartments in which they belong, charges on metabolites etc. Another collection is maintained which is of reactions from the model, the associated reactants and products. Reactions also has additional associated parameters such as subsystem, equation and kineticLaw objects such as flux value of reaction, flux lower bound and upper bound etc. In addition list of all compartments present in the model is formed. The structure/attributes of SBML changes according to the SBML version used.

7.2 Graph Generation

Biological models may not have all attributes present, since some of the parameters are optional such as KineticLaw, ListOfProducts or ListOfModifiers etc. More details of mandatory and optional parameters can be found at sbml.org [30]. In parsed data, information about all the reactions and their relationship with metabolites are gathered. However there is no direct association available from metabolite list to reactions. Nave approach is to iterate through entire reaction list every time when the interest lies in finding all the reactions associated with a metabolite. However this approach is not efficient since at each node, there would be R iterations (R = total no of Reactions). Therefore we have used a hash table (KeyValuePair) where Key is a metabolite M and Value is all the reactions

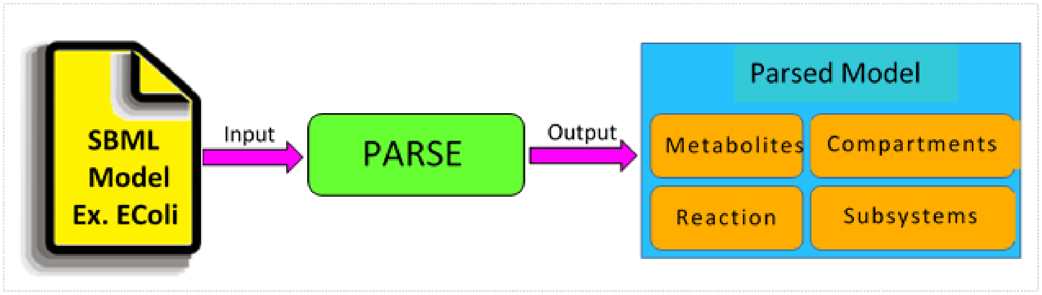


Figure 7.1: Block diagram of parsing module where it takes SBML model as an input and produces model in the form of metabolites, reactions, compartment and subsystems

taking M as a reactant. Similarly a separate hash table is maintained for the entire reaction set to search in constant time bound. Considering reaction as an edge and metabolite as a node would add complexity to the graph since a reaction can take multiple metabolites as an input and produced multiple other metabolites. Therefore the edge would be acting as a hyper-edge connected to many nodes. For simplicity this system can be converted into standard graph where each metabolite or reaction is treated as a node. However these are different types of nodes. In addition node of type reaction also stores additional information with it such as Flux Value, Genes binding etc. Figure 7.2 represents the detailed conversion of hypergraph to Standard Graph.

7.3 Network/Graph Pruning

Cofactors are nonprotein enzymes and also termed as species in the SBML. These enzymes takes part into a chemical transformation, catalyst as an input or byproducts as an output. Examples of cofactors are H2O, CO2, H etc. Given a model, cofactors are high in numbers. Therefore when a network of model is formed, these are high degree nodes, called as hubs.

While searching through the network, these nodes add complexity in the search process. They add invalid pathways during computation. Therefore it is essential to discard the cofactors for the search process. Therefore the network is formed avoiding the cofactors and their edges to the reaction nodes. The cofactor data is found on many online sources. This data is in the form of list of cofactors found in the biological network. During the

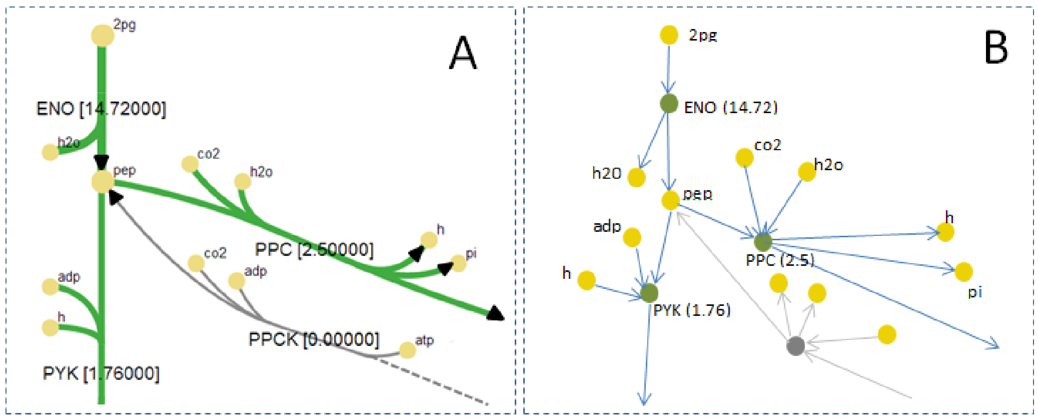


Figure 7.2: Part A - hypergraph with different transformations and pathways. Yellow circles represent the metabolites and green edges are reactions. Arrows indicate the direction of reactions, the metabolites near outgoing arrows are output metabolites (products) of that reaction and metabolites on the opposite side of reaction are input metabolites (reactants). It can be seen that one metabolite is given as input to multiple reactions and multiple reactions can produce the same metabolite as an output. Part B is a conversion from hypergraph to standard graph where each metabolite and reaction is treated as nodes and the relationship between them is considered as an edge between them. We maintain the direction of edges and the weight of the reaction as it is. Analysis: metabolites h2o, co2, h etc. are cofactors, the gray line indicates that weight of the reaction (flux) is 0. Figure A is a snapshot extracted from model Escherichia coli which is represented on our visualization tool PathwayPioneer.com

network formation method, the list is applied for ignoring the cofactor species. Hence reducing the complexity and size of the network as illustrated in Figure 7.3 (A) shows the pruning method on a network and 7.3 (B) shows the resulted pruned graph. Future studies would include cofactors defined based on the degree of a metabolite nodes. If the degree is higher than average degree of all metabolites, it could be labeled as a cofactor.

7.4 Informed Search

The pruned network obtained by removing cofactors and edges is used for search retrieval. Through informed search, early termination can be applied on the network based on network properties. These network properties are structural properties and/or behavioral properties. The DFS with informed search takes one or more properties in combination to terminate the pathways during the. In this project following structural network properties are used: Compartments of metabolites, Subsystem of reactions and behavioral property of

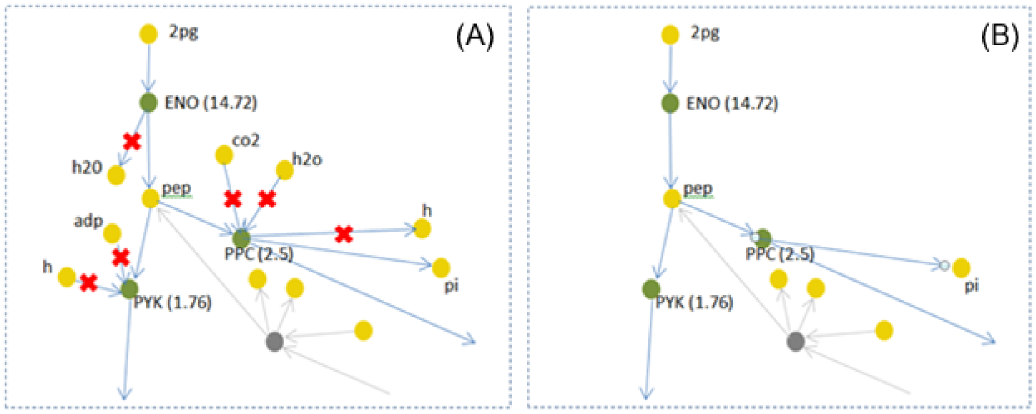
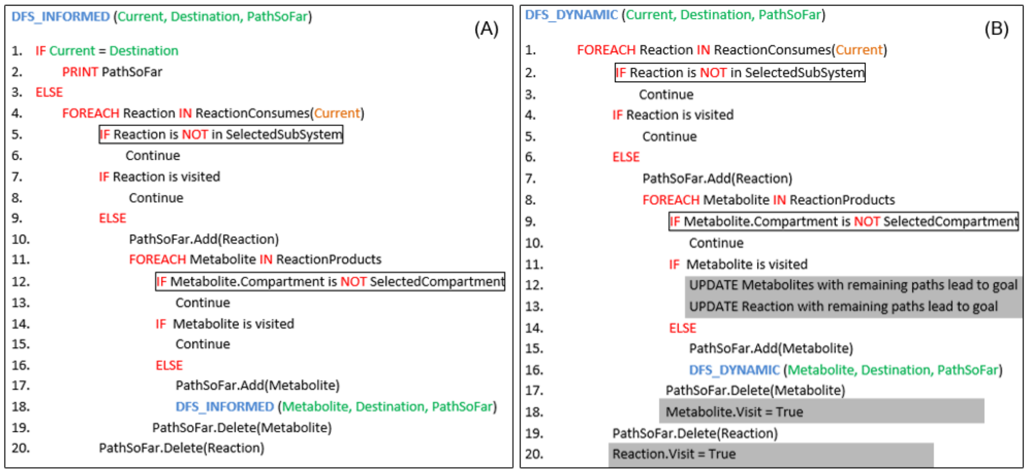


Figure 7.3: Pruning method where we remove the cofactors from the graph; (A): The original graph including cofactors. (B): The graph created after removal of cofactors. In such a small network, we have removed 6 nodes out of 16 nodes, which is very high number for any network, especially when search is to be performed. E Coli Textbook model has 92 metabolite nodes. When all cofactors are removed from this network, it ended up with 45 metabolite nodes, which is near to the half of the total metabolites. To conclude cofactor removal reduces the size of the network significantly and therefore search can be performed faster in such a network than original network.

network is the flux value of reaction.

Uninformed DFS takes source metabolite and destination metabolite as an input. It starts scanning in exhaustive manner and returns all the pathways which exist between them. DFS with dynamic programming stores the intermittent results at the node, for better memory utilization, it stores the reference of next node which leads to the destination metabolite. In the metabolic network, it is possible that nodes as well as reactions could be common for different pathways; therefore storing the results at the intermediate node will cut down many of the additional unsuccessful pathways.

In 3rd algorithm, network properties are customized by user. Selected compartment or subsystem for search and flux values associated with a reaction. If the reaction flux is 0 the leading pathway could be avoided. Figure 7.4 illustrates the snapshot of the algorithm for Informed DFS with Compartment and Subsystem setting.



(A)

```
DFS_INFORMED (Current, Destination, PathSoFar)

IF Current = Destination
    PRINT PathSoFar
ELSE
    FOREACH Reaction IN ReactionConsumes(Current)
        IF Reaction is NOT in SelectedSubSystem
            Continue
        IF Reaction is visited
            Continue
        ELSE
            PathSoFar.Add(Reaction)
            FOREACH Metabolite IN ReactionProducts
                IF Metabolite.Compartment is NOT SelectedCompartment
                    Continue
                IF  Metabolite is visited
                    Continue
                ELSE
                    PathSoFar.Add(Metabolite)
                    DFS_INFORMED (Metabolite, Destination, PathSoFar)
                PathSoFar.Delete(Metabolite)
        PathSoFar.Delete(Reaction)
```

(B)

```
DFS_DYNAMIC (Current, Destination, PathSoFar)

FOREACH Reaction IN ReactionConsumes(Current)
    IF Reaction is NOT in SelectedSubSystem
        Continue
    IF Reaction is visited
        Continue
    ELSE
        PathSoFar.Add(Reaction)
        FOREACH Metabolite IN ReactionProducts
            IF Metabolite.Compartment is NOT SelectedCompartment
                Continue
            IF  Metabolite is visited
                UPDATE Metabolites with remaining paths lead to goal
                UPDATE Reaction with remaining paths lead to goal
            ELSE
                PathSoFar.Add(Metabolite)
                DFS_DYNAMIC (Metabolite, Destination, PathSoFar)
            PathSoFar.Delete(Metabolite)
            Metabolite.Visit = True
    PathSoFar.Delete(Reaction)
    Reaction.Visit = True
```

Figure 7.4: (A): Informed DFS branch and bound based on Compartment of metabolite and Subsystem of the reaction; (B): Present DFS Dynamic algorithm with branch and bound

CHAPTER 8
EVALUATION AND RESULTS

8.1 Results

We have tested all of our implemented algorithms and logged various parameters for the analysis. Starting with the SBML Parser, it is compared with a well-known and widely used SBML parser libSBML. Figure 8.1 shows the comparison of libSBML and our SBML Parser. The test is conducted on different biological models with varying sizes (reactions and metabolites). Graph also specify the total number of nodes each model has.

All of the implemented search algorithms are compared based on different logged parameters. In Figure 8.2, the line graph is plotted for the comparison of total number of searches performed by each algorithm in a given time range. The second part of Figure 8.2 shows the comparison based on the total number of pathways extracted by each algorithm with respect to time in seconds. The figure description explains the graph and detailed observations. Figure 8.3 presents a heat map from a variable which is ratio of total pathways found against total number of searches done. It shows that dynamic programming works better than other algorithms and finds maximum pathways with maximum possible elimination. The DFS with compartment and DFS with subsystem perform better than uninformed DFS.

8.2 Future Work

The tool finds pathways between two metabolites using informed DFS search. In the future this could be extended to find pathways between the set of metabolites. Many of the reactions take more than one metabolites as a reactant, therefore this would be interesting to see how pathways would be reduced by searching set of metabolites as an input and set

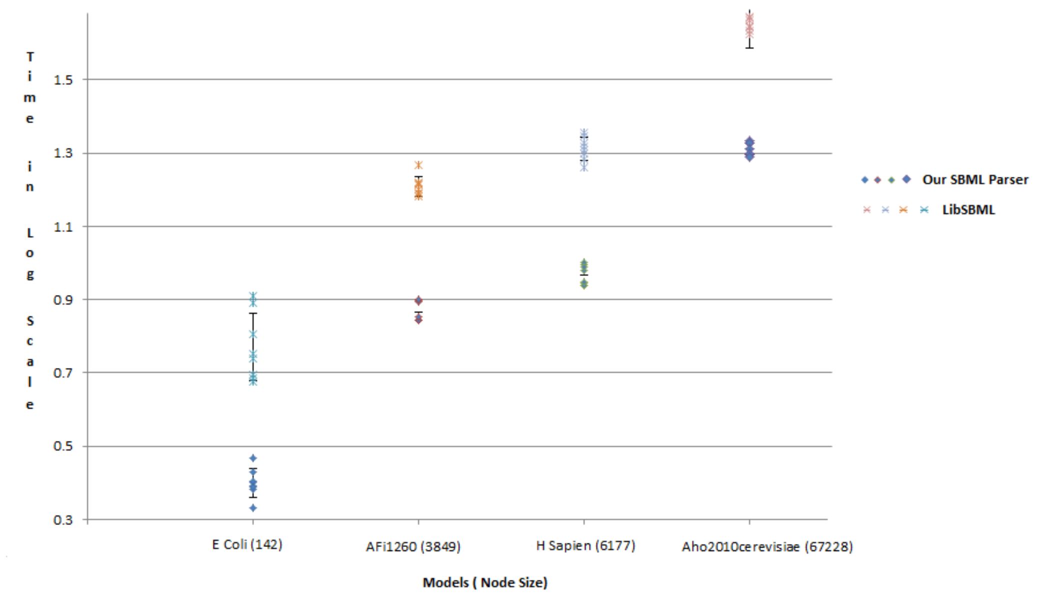


Figure 8.1: Comparison of libSBML and Our SBML Parser. The horizontal axis represents the different models for which the parser is tested. Both of the libraries are tested on the same configured system with same set of inputs. The code written for both the parser is in C# and tested on the same Microsoft Windows 7 system with same hardware configuration. We have taken samples of each model for 10 times for both the parsers and plotted a standard deviation and mean graph. The time on Y axis is log of base 200.

of metabolites as an output. Further the set of metabolites given as an input or output could be treated as only one compound.

Currently the graph is pruned based on cofactors, which is a list, however it would be interesting to understand how the cofactors are defined and selected dynamically. In addition if all the important metabolites are defined in a pathway, preprocessing can be done on this graph and heuristics function may be applied for faster retrieval.

It would be also important to define important pathways for users. A further study can be done to understand importance of a specific pathway for biologist and it would be given preference over other pathways.

Similar to our observation, Küffner [19] pointed out that total number of pathways in an unrestricted network is a very large set. Therefore it would be necessary to find only those pathways which have biological importance. One of the future studies would be to find all the valid pathways on the KEGG database for a given metabolite and compare

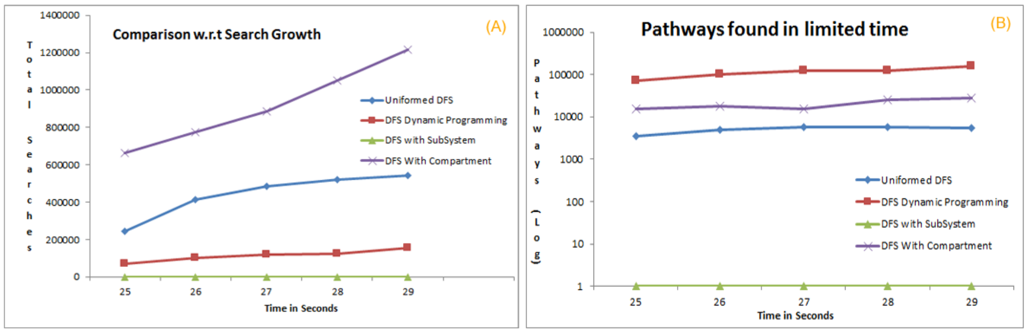


Figure 8.2: The comparison of search algorithms based on total searches (A) performed by Uninformed DFS, DFS with Dynamic Programming, DFS with Subsystems (DFSS) and DFS with Compartment. The algorithms are executed for 25-30 seconds time range and their respective results at that point are noted. DFS with compartment (DFSC) performs maximum searches compared to other algorithms. From the fact that DFSC terminates a pathway as soon as it reaches to a metabolite from another compartment, it performs more searches than uninformed DFS. Although dynamic programming searches exhaustively in the beginning phase, it keeps storing intermediate results at the metabolites and reactions. Therefore after exhaustive beginning search phase, it only adds the pathways which leads to destination (and stored in cache). Therefore the pathways found through dynamic programming are more than those from DFSC, however total number of searches are less. (B): The line graph shows the comparison of search algorithms for number of pathways found in a given time range. Dynamic programming performs better than any other algorithm since it uses results stored in the intermediate cache and therefore it searches pathways quickly. In both the figures DFSS lies on the bottom of the graph. For a given subsystem, the DFSS eliminates large number of reactions which lies in the other subsystems. Therefore it finds only those pathways which are formed by all those reactions which are part of given subsystem and performs very limited searches within the subsystem. (The number of pathways are much lesser than the total number of pathways found by other algorithms.)

those in current SBML model. Finding the shortest pathways could also limit the search on such a network and return only those pathways which have k number of reactions etc.

Currently pathways are represented in the form of tabular data shown in Figure 8.4. However it would be more interesting to embed this algorithm with visualization. Current available tools like PathwayPioneer [30] can be used to visualize the metabolic network which is in the form of SBML. Therefore visualization tools can be extended to visualize the pathways which are returned by our algorithm. In PathwayPioneer a user can focus on a particular subsystem of a model and we found that pathway search in subsystem of model works faster than unrestricted search and returns fewer results as compare to the pathways in a whole system. Therefore set of metabolites and pathways returned by our algorithm

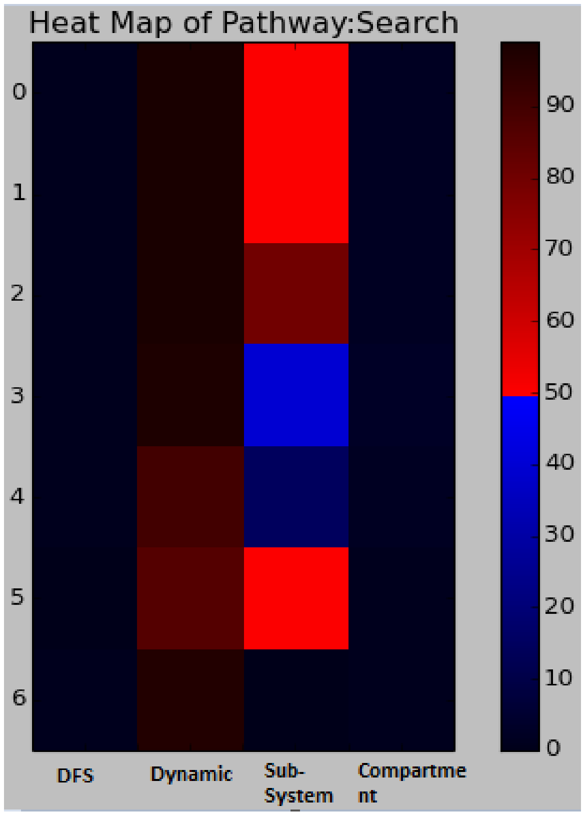
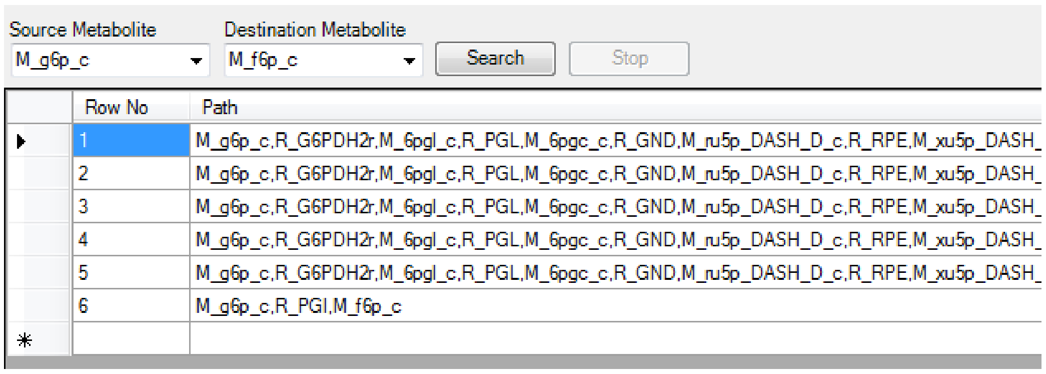


Figure 8.3: The variable used in the heat map is the ratio of total number of pathways found with respect to total number of searches in the given time. We found that DFS Dynamic performs far better than other search algorithms. It finds maximum pathways within close range of total searches in given time.

can be highlighted using some different color combinations to show the results.



Source Metabolite: M_g6p_c

Destination Metabolite: M_f6p_c

Row No	Path
1	M_g6p_c,R_G6PDH2r,M_6pgl_c,R_PGL,M_6pgc_c,R_GND,M_ru5p_DASH_D_c,R_RPE,M_xu5p_DASH_
2	M_g6p_c,R_G6PDH2r,M_6pgl_c,R_PGL,M_6pgc_c,R_GND,M_ru5p_DASH_D_c,R_RPE,M_xu5p_DASH_
3	M_g6p_c,R_G6PDH2r,M_6pgl_c,R_PGL,M_6pgc_c,R_GND,M_ru5p_DASH_D_c,R_RPE,M_xu5p_DASH_
4	M_g6p_c,R_G6PDH2r,M_6pgl_c,R_PGL,M_6pgc_c,R_GND,M_ru5p_DASH_D_c,R_RPE,M_xu5p_DASH_
5	M_g6p_c,R_G6PDH2r,M_6pgl_c,R_PGL,M_6pgc_c,R_GND,M_ru5p_DASH_D_c,R_RPE,M_xu5p_DASH_
6	M_g6p_c,R_PGI,M_f6p_c

Figure 8.4: Screenshot of GUI used for current pathways representations.

CHAPTER 9
CONCLUSIONS

Architecture of PathwayPioneer is designed in such a way that it can scale very well with respect to large user base. Through Markov testing model, our tool is tested for 50 concurrent users randomly performing different operations on PathwayPioneer. PathwayPioneer was able to handle 50 concurrent user with acceptable average response time. Through MVC pattern, the tool was able to filter the valid URLs, modularized the code and make the application light weight. We implemented Python service which allows python engine to run using native interpreter. Python engine was able to serve multiple requests in parallel while responding back with standard IO to webserver.

In Pathway Search in Metabolic Network, we have implemented various informed search algorithms for early termination. We have also implemented uninformed Depth First search to compare the results of informed searches and uninformed searches in biological network. Graph representation of a metabolic network is a hyper-graph having hyper edges. When converted it to the standard graph it poses the properties like directed edges, weights to some nodes and cycles. The algorithms we have implemented utilize the structural and behavioral properties of this graph and it eliminates many pathways during its early retrieval. The elimination of Cofactors prunes the graph very well and size of the graph is reduced. Overall the DFS with dynamic programming performs better than other algorithms. DFS with subsytem performs faster than other algorithms, however it returns limited results only from the selected subgraph.

SBML parser implemented in this project, performs better than existing SBML parser (libSBML) which is implemented in C#.

REFERENCES

[1] A. Ebrahim, J. A. Lerman, B. O. Palsson, and D. R. Hyduke, "Cobrapy: constraints-based reconstruction and analysis for python," *BMC Systems Biology*, vol. 7, p. 74, 2013.

[2] M. Foord and C. Muirhead, *IronPython in action.* Greenwich CT: Manning Publications Co., 2009.

[3] B. lloyd. Python for .net. [Online]. Available: http://pythonnet.sourceforge.net/

[4] Enthought Python Distribution for NUMPY, SCIPY for IRONPYTHON. [Online]. Available: http://blog.enthought.com/python/scipy-for-net/

[5] IronPython Documentation by Microsoft. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc300810.aspx

[6] library which allows IronPython to import and use compiled CPython extensions. [Online]. Available: http://code.google.com/p/ironclad

[7] Python for .net tool documentation http://pythonnet.sourceforge.net/readme.html. [Online]. Available: http://pythonnet.sourceforge.net/readme.html

[8] SBML visualization tools http://sbml.org/SBML_Software_Guide/SBML_Software_Summary#cat_11. [Online]. Available: http://sbml.org/SBML_Software_Guide/SBML_Software_Summary#cat_11

[9] SBML analysis tools http://sbml.org/SBML_Software_Guide/SBML_Software_Summary#cat_1. [Online]. Available: http://sbml.org/SBML_Software_Guide/SBML_Software_Summary#cat_1

[10] A. C. Villéger, S. R. Pettifer, and D. B. Kell, "Arcadia: a visualization tool for metabolic pathways," *Bioinformatics*, vol. 26, no. 11, pp. 1470–1471, 2010.

[11] J. Boele, B. G. Olivier, and B. Teusink, "Fame, the flux analysis and modeling environment," *BMC Systems Biology*, vol. 6, no. 1, p. 8, 2012.

[12] S. I. Berger, R. Iyengar, and A. Maayan, "Avis: Ajax viewer of interactive signaling networks," *Bioinformatics*, vol. 23, no. 20, pp. 2803–2805, 2007.

[13] Model/View/Controller. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff649643.aspx

[14] MVC VS N-Tiers - MSDN Microsoft. [Online]. Available: http://social.msdn.microsoft.com/Forums/en-US/97562908-f75b-4afb-9906-91a6b19a28bb/mvc-vs-ntiers

[15] WATIN: Web Application Toolkit In .Net. [Online]. Available: http://watin.org/

[16] Selenium browser automation. [Online]. Available: http://docs.seleniumhq.org/

[17] SBML Metabolites information. [Online]. Available: http://en.wikipedia.org/wiki/Metabolites

[18] CoFactors information. [Online]. Available: http://academic.brooklyn.cuny.edu/biology/bio4fv/page/coenzy_.htm

[19] D. C. McShan, S. Rao, and I. Shah, "Pathminer: predicting metabolic pathways by heuristic search," *Bioinformatics*, vol. 19, no. 13, pp. 1692–1698, 2003.

[20] KEGG Database online. [Online]. Available: http://www.genome.jp/kegg/

[21] R. Küffner, R. Zimmer, and T. Lengauer, "Pathway analysis in metabolic databases via differential metabolic display (dmd)," *Bioinformatics*, vol. 16, no. 9, pp. 825–836, 2000.

[22] W. Zhou and L. Nakhleh, "Properties of metabolic graphs: biological organization or representation artifacts?" *BMC Bioinformatics*, vol. 12, no. 1, p. 132, 2011.

[23] K. Faust, P. Dupont, J. Callut, and J. Van Helden, “Pathway discovery in metabolic networks by subgraph extraction,” *Bioinformatics*, vol. 26, no. 9, pp. 1211–1218, 2010.

[24] K. Faust, D. Croes, and J. van Helden, “Prediction of metabolic pathways from genome-scale metabolic networks,” *Biosystems*, vol. 105, no. 2, pp. 109–121, 2011.

[25] D. Croes, F. Couche, S. J. Wodak, and J. Van Helden, “Metabolic pathfinding: inferring relevant pathways in biochemical networks,” *Nucleic Acids Research*, vol. 33, no. suppl 2, pp. W326–W330, 2005.

[26] D. Croes, F. Couche, S. J. Wodak, and J. van Helden, “Inferring meaningful pathways in weighted metabolic networks,” *Journal of Molecular Biology*, vol. 356, no. 1, pp. 222–236, 2006.

[27] A. Mithani, G. M. Preston, and J. Hein, “Rahnuma: hypergraph-based tool for metabolic pathway prediction and network comparison,” *Bioinformatics*, vol. 25, no. 14, pp. 1831–1832, 2009.

[28] BioCyc online tool for metabolic route search. [Online]. Available: http://biocyc.org/meteng?organism=ECOLI

[29] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden *et al.*, “The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models,” *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.

[30] PathwayPioneer.com online tool for Pathway Visualization and Flux Analysis. [Online]. Available: http://www.pathwaypioneer.com