Solving Classical Problems in Computational Molecular Biology Using Distributed Computing and Web Technologies

(degas.js – **d**istributed **e**volutionary (**g**enetic) **a**lgorithm **s**panned)

Carleton University, School of Computer Science
Honours project (COMP4905) by Rakhim Davletkaliyev
Supervisor: Evangelos Kranakis
August 22nd, 2011

Table of Contents

List of Figures

Abstract

One of the most promising ways to solve the hardest computational problems nowadays is distributed and parallel computing. The motivation is easy: if problem is too big and hard, divide it into pieces and let several entities solve their piece; then, consolidate the results into the potential answer.

Most of personal computers and computing devices today are connected to internet and have web-browser installed, which usually allows to run code on host machine. Yet, this feature is not widely used as a tool to divide and distribute computation.

The goal of this project is to create a JavaScript framework that would allow anyone to set a server and a webpage, connecting to which a user's web-browser will contribute to overall computation. For example, the famous traveling salesman problem can has plenty of potential solutions.

With help of degas.js (**d**istributed **e**volutionary (**g**enetic) **a**lgorithm **s**panned framework) one can define this problem on the server (as array of coordinate points which salesman must visit), define the way to evaluate a solution (as function summing the distance travelled) on the client side. Now, every user that visits the webpage contributes their CPU cycles to generate and send potential solutions to the server.

0.0 Motivation

Many problems in computational molecular biology (as well as many problems in computer science and computer science related fields in general) are hard or impossible to solve sequentially; many of them are NP-hard or NP-complete. With help of parallel and/or distributed computing, some of these problems can be dealt with up to some degree of correctness.

There've been several attempts to create volunteer-based distributed computing projects to perform simulations of protein folding (Folding@Home), analyze radio signals in search of extraterrestrial life (SETI@Home) and others. Most of them require specific software to be installed on each peer's computer, which takes unused (or scheduled by user) CPU time to contribute to overall computation. Still, many of these programs experience financial difficulties and lack of attention both from general society and scientific community.

The idea of this project is to create similar infrastructure for volunteer-based computation for simpler and more fundamental classical problems in computer science, which can be converted to corresponding problems in computational molecular biology. But the role of client-side software is to be played by volunteer's web browser with help of web-oriented, dynamic and powerful language of JavaScript (ECMAScript). Most of the modern web-browsers have JavaScript enabled by default, and most of the modern mobile devices' browsers as well, which means contributions to overall computation can be done from desktop computer, tablets, mobile phones, etc.: the global internet population is over 2,000,000,000 people!

While overall generic goal is quite vague, great number of limitations provided by the web and browser environment arose during the development, narrowing the task and toolset.

0.1 Generic structure of the future framework

The generic structure and corresponding requirements of the program parts on the initial stage before development were the following:

- **Server**
  - Must provide the tools to divide the problem
  - Must serve data to clients
  - Must be scalable

- **Client**
  - Must provide the tools to define solution to the problem and do the computation to generate a solution
  - Must work seamlessly, without distracting the host system
  - Must be as autonomous as possible

- **Server-client connection**
  - Must allow asynchronous data transfer
  - Must allow large amounts of data to be transferred

1.0 Existing research and solutions

Having this in mind, we should first evaluate the currently available solutions and works in related fields. As was said in the previous chapter, distributed computing via web is not new and it is quite possible that the structure described here is not fully possible to implement while saving appropriate amount of resources.

1.0.1 JSDC http://jsdc.appspot.com/

Javascript Distributed Computing (JSDC) is a small cross-domain friendly, simple 570 byte snippet of javascript code. It's just a prove of concept that such distributed computation is possible with browser-enabled javascript. It currently does one of the most easily distributed tasks: reverse hashing. Each client is assigned a work unit where it searches from xxxxaa to xxxxzz and sees if any matches occur. The data is sent to a remote server which logs it and assigns a new work unit.

1.0.2 Ravan http://www.andlabs.org/tools/ravan.html

Ravan is a JavaScript based Distributed Computing system that can perform brute force attacks on salted hashes by distributing the task across several browsers. It makes use of HTML5 WebWorkers to start background JavaScript threads in the browsers of the workers, each worker computes a part of the hash cracking activity.

As seen on figure 1, the structure of Ravan is very similar to proposed one. The server distributes the initial data among workers and awaits for results. The way creators of Ravan satisfied the requirement 2 of the client (Must work seamlessly, without distracting the system) is with help of new, not yet standardized tool called Web Workers.

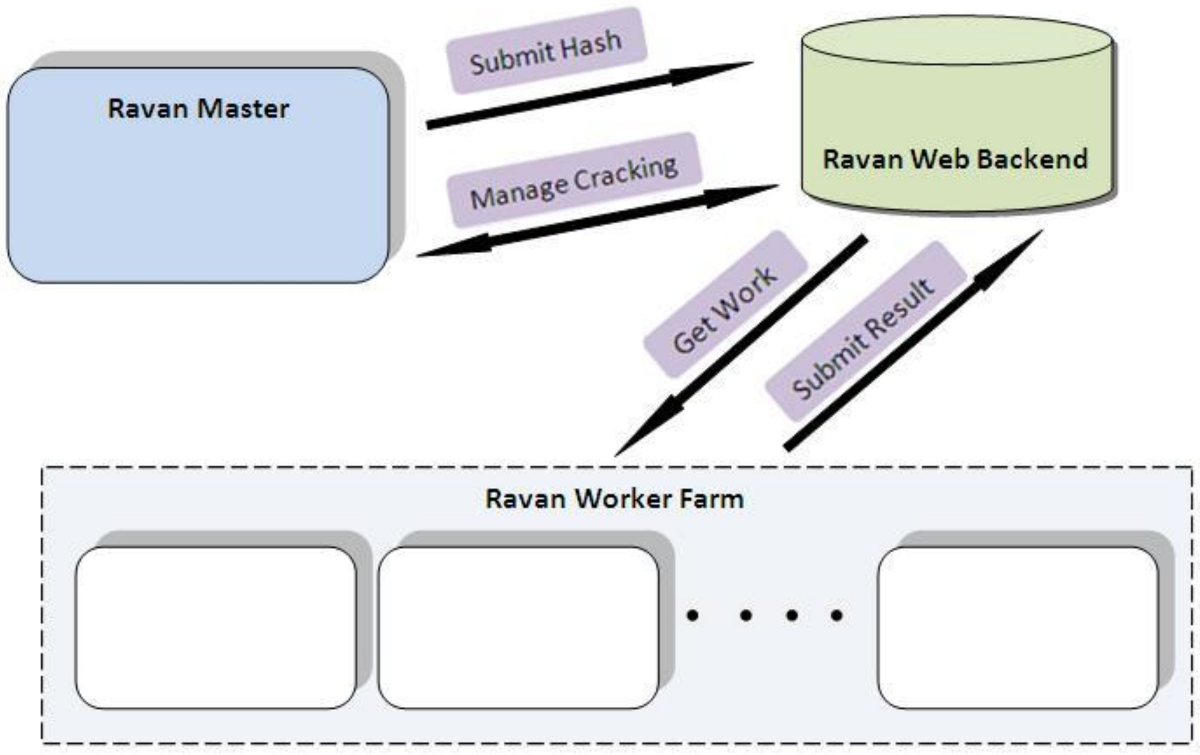


Figure 1. Structure of Ravan

Web Workers define an API for running scripts, basically JavaScript, in the background independently of any user interface scripts. This allows for long-running scripts that are not interrupted by scripts that respond to clicks or other user interactions, and allows long tasks to be executed without yielding to keep the page responsive.

Even though not all browsers with JavaScript enabled have web worker implemented (and even among those – not always web worker feature is enabled!), web workers have great potential and will be widely used soon enough.

Usually, JavaScript on the page takes few percents of time to do its computation, while most of the time it waits for user interface and DOM-related tasks to be done. This is a huge under-optimization. Web workers allow to run usually highly efficient and fast JavaScript code in a separate (parallel to UI thread) thread. It allows for the browser to continue with normal operation while running in the background. Web Worker specification is a separate specification from HTML5 specification but can be used with HTML5.

1.0.3 MapRejuice https://github.com/ryanmcgrath/maprejuice

MapRejuice is a distributed MapReduce implementation for JavaScript. MapReduce is a software framework introduced by Google in 2004 to support distributed computing on large data sets on clusters of computers. The framework is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as their original forms: “map” function divides the problem into pieces for distribution among workers; “reduce” function generates a solution to a piece of the problem.

1.0.4 Cyberaide JavaScript: A JavaScript Commodity Grid Kit

This paper describes a service oriented architecture and Grid abstraction framework that allows us to access Grids (large computational entities) through JavaScript. Obviously, such a framework integrates well with other Web 2.0 technologies. The framework consists of two parts: a client Application Programming Interface (API) to access the Grid via JavaScript and a mediator service and API through which the Grid access is channeled.

The framework uses commodity Web service standards and provides extended functionality such as asynchronous task management, file transfer, and workflow management based on our previous work. The availability of our framework simplifies not only the development of new services, but also the development of advanced client side Grid applications that can be accessed through Web browsers.

1.0.5 Asynchronous distributed genetic algorithms with JavaScript and JSON

This paper presents a distributed evolutionary computation system that uses the computational capabilities of the web browser. Asynchronous JavaScript and JSON (JavaScript object notation, a serialization protocol) allow anybody with a Web browser to participate in a genetic algorithm experiment with little effort, or none at all. Paper explores the performance of this kind of virtual computer by solving simple problems such as the royal road function and analyzing how many machines and evaluations it yields.

It also examines possible performance bottlenecks and how to solve them, and, finally, issue some advice on how to set up this kind of experiments to maximize turnout and, thus, performance. The experiments show that it is possible to obtain high, and to a certain point, reliable performance from volunteer computing based on AJAJ, with speedups of up to several (averaged) machines.

This particular paper alone has enough proof of concept than any other implementation mentioned.

1.0.6 Generic workers: towards unified distributed and parallel JavaScript programming model

This paper introduces *generic workers*, a programming model for JavaScript unifying parallel and distributed computing paradigms that allows the same application to run well on a variety of clients while utilizing the available resources in the best possible way. It describes the design and implementation of an infrastructure supporting our programming model and evaluates performance of selected applications running on devices with differing computational capabilities.

1.1 Findings

Examining existing work and solutions lead to the following generic conclusions that helped to narrow to goal for the project:

- Genetic algorithm (evolutionary computation) is greatly distributable task by its nature.

- Genetic algorithms are easily adjustable to fit the resource allowance of host machine

- Slow network interaction is natural bottleneck for the entire class of problems

- Modern JavaScript engines in web browsers allow somewhat heavy tasks to be executed in browser execution thread (assuming that thread itself has separate event-driven (non-blocking) sub threads for user interface/rendering and JavaScript code.

- Pretty satisfying results were found in early 2009, two and a half years ago. These two and a half years brought more optimized web-browsers, JavaScript engines and evolution of promising technologies like web workers and web sockets.

- Web Worker is the best way to allow somewhat heavy computation without blocking the user interface thread of a web browser.

1.2 Choosing evolutionary way

At this point the decision was made to use evolutionary computation as the primary tool for solving problems in distributed fashion.

Evolutionary computation is a subfield of artificial intelligence (more particularly computational intelligence) that involves combinatorial optimization problems. Evolutionary computation uses iterative progress, such as growth or development in a population. This population is then selected in a guided random search using parallel processing to achieve the desired end. Such processes are often inspired by biological mechanisms of evolution. The solution to a given problem is literally evolved from random or bad solutions by mutating their parts and combining them (crossover) in iterative way.

This choice, of course, limits the ability, as not any problem is easily solvable by genetic algorithms. But it is important to keep in mind the natural limit for the entire concept: not all problems are easily distributable (dividable) at all! Interestingly enough (yet, expectable): if problem is easy to parallelize, then it is possible to evolve solutions in genetic way.

Parallel implementations of genetic algorithms come in two flavors:

- Coarse-grained parallel genetic algorithms assume a population on each of the computer nodes and possibly migration of individuals among the nodes.

- Fine-grained parallel genetic algorithms assume an individual on each processor node which acts with neighboring individuals for selection and reproduction.

First way fits our generic requirements better, because local population can continue evolving on its own, regardless of internet connection and server responses in general.

Second flavor, on the other hand, depends entirely on healthy network, allowing constant flow of data between nodes and server. Consequently, it puts more pressure on network channels, the natural bottleneck of any distributed problem.

1.3 Updated structure of the future framework

Findings mentioned in the previous paragraph lead to the more specific and detailed description and requirements for the future framework:

- **Server**
  - Must provide the tools to divide the problem
  - Must be able to verify received solutions (individuals)
  - Must serve data to clients
  - Must be extendable
  - Must be scalable

- **Client**
  - Must provide the tools to define solution to the problem and do the evolutionary computation to generate a solution
  - Must provide ways to adjust the heaviness of genetic computation
  - Must be extendable
  - Must work seamlessly, without distracting the host system
  - Must be as “quiet” on the network as possible
  - Must be as autonomous as possible

- **Server-client connection**
  - Must allow full-duplex data transfer
  - Must allow large amounts of data to be transferred

2.0 Development

The following chapters will describe development process, choices and sacrifices and also will include information needed and steps to build a simple application using degas framework.

2.1 Client

Client side consists of a webpage and a single web worker which consequently builds next generations of population of solutions while exchanging some data with server when possible. Client should be able to continue evolving without being connected to server and should detect when such connection is available to exchange data.

Data to be received by a client from the server can be:

- Initial data. Whatever needed for evolution to start. Some problems require input data (for example, traveling salesman: points is initial data), some problems don't (for example, finding prime numbers). Note that data necessary for evaluating a solution (fitness function) is not considered initial, as it is static and doesn't change. Initial data is problem-specific information for a particular implementation of problem.

- Global data. Some information about overall computation, used resources, leader board of hosts which generated a solution with highest fitness, etc. This data is not essential to computation and has lower priority over the network interface.

2.1.1 Classes. Individuals.

Pure JavaScript provides somewhat clumsy way to create objects with certain fields and methods. To take advantage of prototypal nature of JavaScript and create an easy interface for class and object handling, the first layer of degas framework was developed; it is a simple function called "Class" that makes it easy to create classes, objects of that class, create classes by inheriting properties from another object and even embed one object into another.

As an example of using Class function we will define an Individual object. Individual object must contain the genetic sequence that represents potential solution, a fitness function for evaluating genetic sequence and some other fields and methods needed for Individual to exist and participate in evolution. The following code is simplified version of code that used in degas.js:

```
var Individual = degas.Class({
     initialize: function (sequenceLength, bitsPerCell) {
          this.sequence = new Array(sequenceLength);
          this.fitness = 0;
     },

     fitness: function() {
          // returns fitness as numeric value
     }...
});

var myInd = new Individual(100, 3);
```

First part describes a class called Individual that takes two parameters to constructor: length of genetic sequence of cells and size of each cell. The last line creates a new variable myInd as instance of Individual object with sequence length of 100 cells; each cell is 3 bit long.

This approach of using sequence of cells was chosen to allow any type of data to operate on. For example, if solution to the problem can be described as a binary string, then it will make most sense to create individual's cell size to 1 bit: this way the entire sequence will consist of X cells which are bits.

If, on the other hand, the solution is in UTF-32 (USC-4) characters, then each cell should be 32 bit long (length of UTF-32 character).

By default, degas framework considers cells atomic structures, meaning that any modification of the sequence (mutation or crossover) affects the sequence of cells only, not cells themselves. This is done for safety and probabilistic consistency, since not all alphabets can be tightly described in bits without leaving unused combinations. For example, if alphabet of solution consists of three characters, one bit is not enough to represent a letter, but two bits are too much: they provide four combinations.

So, one untaken combination may be used to represent some letter second time (which contributes to inconsistency in probabilities and therefore lower entropy; this can be harmful to genetic algorithm's run). Alternatively, untaken combination can be left unused. In this case if mutation or crossover takes places on bit level, unused combinations should be checked for, which will take away from useful computation.

Of course, proper use of Huffman coding would be the best way to avoid this issue altogether. Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.

But for current release the framework assumes cells to be atomic indivisible structures by default. Of course, since this is an easily extendable framework, it is easy to provide a bit-level functionality.

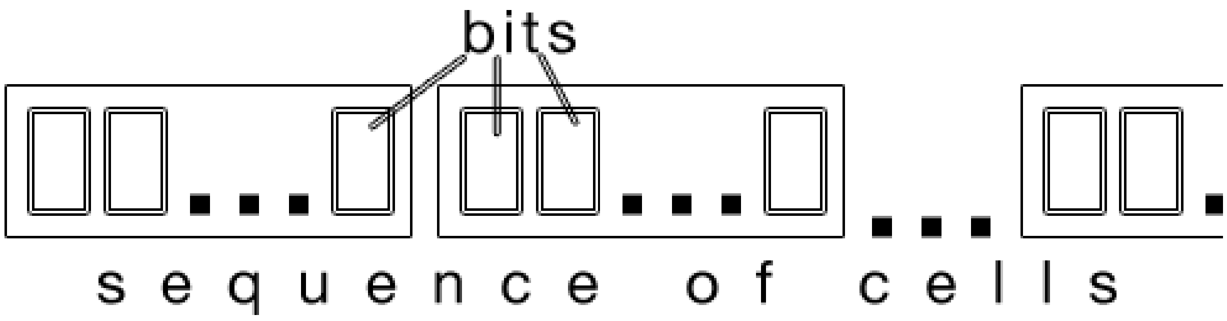


Figure 2. Structure of individual genetic sequence.

Degas.js allows to create new objects that inherit from another objects using the same function "Class":

```
var SuperIndividual = degas.Class(Individual, {
     initialize: function(){
          this.$super('initialize', arguments);
          this.color = 'green';
     },

     currentColor: function() {
          return this.color;
     }
});
```

In this code snippet a new class is described that inherits from old Individual class, introduces a new field "color" and new method that returns color. Class SuperIndividual has all fields and methods of its parent object.

This feature is particularly interesting, because it allows to dynamically define new classes at run time. For example, if evolution has started and server sends new information to be included in individual's description. Or, server sent an individual from another host which has slightly different structure/implementation.

The Class function also allows mixins: object embedded within another object. In object-oriented programming languages, a mixin is a class that provides a certain functionality to be inherited or just reused by a subclass, while not meant for instantiation (the generation of objects of that class). Inheriting from a mixin is not a form of specialization but is rather a means of collecting functionality.

```
var MixinIndividual = degas.Class({
     include: Individual,

     initialize: function(type) {
          this.type = type;
     }
});
```

In this code snippet a new class MixinIndividual is created that includes Individual class.

JavaScript does not provide mixin functionality, but it is easy enough to mimic it by copying methods from one object to another at runtime. This is a source of great expressive power when it comes to changing the object dynamically at runtime without stopping the program, recompiling etc.

It is important to take full advantage of dynamic JavaScript and object handling interface described here is believed to do so.

In addition to the most fundamental properties of individual (sequence and fitness function), degas provides several methods by default and (of course) a possibility to extend it.

decodeSequence converts the sequence of cells into a string of characters. decodeSequence expects function decode to exist, and if it doesn't, by default it sums up bits in every cell and returns a string of adjacent bit sums. decode() must be provided by user. Recall example with three characters in the alphabet. decodeSequence would return a string that consists of those characters.

mutate takes two parameters – mutation type and defaultProbability – and mutates individual. If no mutation type is provided the method leaves the original sequence untouched. Mutation types among other constants are stored in degas.consts object.

Each individual has mutationProbability set to false by default. In this case the global parameter degas.config.mutationProbabilityForCell is used. It describes what percent of cells of individual are affected to mutation if individual itself is mutating. The percentage of individuals to mutate is described in degas.config.mutationProbabilityForIndividual. These values are both set to 0.1 (10%) by default. As any other value, it can be changed dynamically during runtime.

2.1.2 Population

Population consists of pool of individuals and some methods to operate over it. Population constructor takes population size, length of individual sequence and number of bits per cell as parameters:

```
var myPopulation = new Population(100, 25, 10);
```

This line of code creates a population of 100 individuals, each having a sequence of 25 cells, each cell is 10 bit long. When new population is created all individuals are initialized with random bits by default. For some problems it would make sense to initialize individuals with some data.

compareIndividuals method acts as a sorting function for comparing two individuals. By default it sorts individuals by fitness only (higher fitness is better), but can be easily replaced with something more complex.

crossoverIndividuals takes two sequences and crossover type as parameters. It creates a new sequence of the same length as parent sequences and performs the crossover operation with two parents. If no crossoverType is provided the child sequence is set to be identical to first parent. Types of crossover among other constants are stored in degas.consts object. Crossover type is set to “Random Split” by default (in degas.config.crossoverType): it splits each parent sequence in the same point and connects two parts into a child sequence.

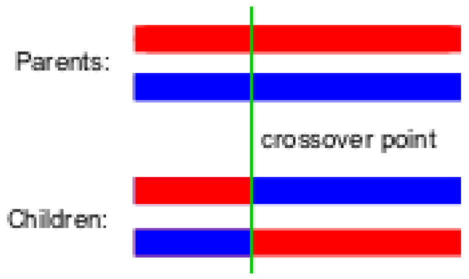


Figure 3. Random split crossover.

Other types available are two point random split:

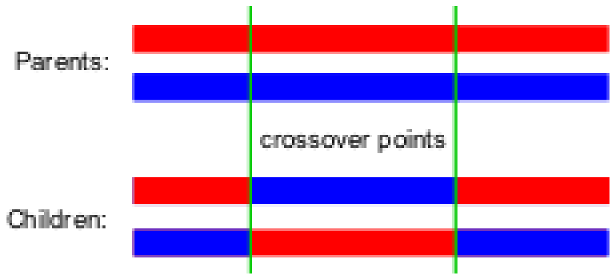


Figure 4. Two point random split.

and uniform crossover:

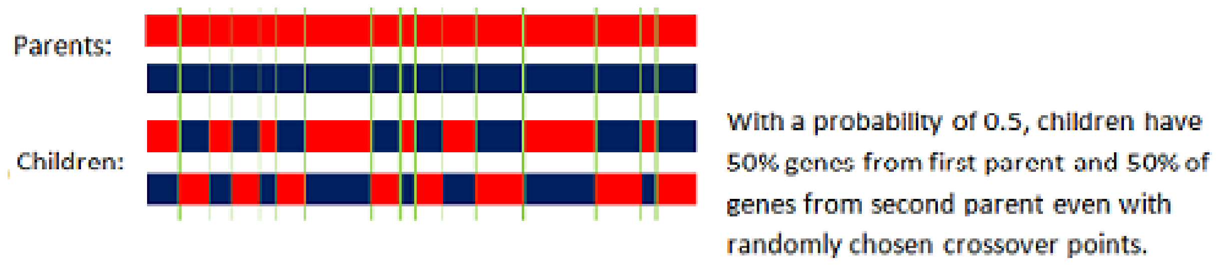

Parents:
Children:
With a probability of 0.5, children have 50% genes from first parent and 50% of genes from second parent even with randomly chosen crossover points.

Figure 5. Uniform crossover.

performCrossover is a generalization of crossoverIndividuals() method. It takes crossover for individual probability as a parameter and performs crossover on entire population. The percentage of individuals to be replaced by children of the best individuals is stored in degas.config.crossoverProbability. This value is set to 0.3 (30%) by default. As any other value, it can be changed dynamically during runtime.

sumUpFitness returns the sum of fitness values of all individuals. Note that this value can by no means describe the level of success of computation as one individual with perfect solution can generate the same sum of fitness values as thousand individuals with bad solutions. Also note, that this function only sums up fitness values stored in individuals, while individual sequences can be already mutated or changed, but value is not yet updated. So, sumUpFitness can be inaccurate by one generation.

computeFitness calls computeFitness() function on every individual, stores returned value in each individual's fitness field and returns the sum. Note that this function does provide the most accurate sum of fitness values of all individuals, but takes more time to execute.

mutate method mutates population. It takes mutation probability for individual and mutation probability for cell as input parameters. These values are stored in degas.config.mutationProbabilityForIndividual and degas.config.mutationProbabilityForCell respectively and both set to 0.1 (10%) by default.

buildNextGeneration performs mutation, crossover, updates fitness and sorts the population using compareIndividuals sorting function. It uses configuration values stored in degas.config. This function is fundamental evolutionary step.

2.1.3 Network interface

Web sockets were chosen as primary channel for data exchange between nodes and server. Degas provides wsClient class for this purpose.

```
var client = new wsClient('182.94.22.12', '8044');
```

This line of code creates and initializes new web socket connection to host 182.94.22.12 and port 8044. When socket is created it automatically sends the greeting message for degas server – a single string "client-connected'. Note that this message is not a replacement for regular handshake data exchange that takes place upon socket creation.

The web socket client then awaits for initial data message from server, after receiving which evolution can start.

wsClient has send method that takes any object, serializes it using JSON format and sends as a sequence of characters to degas web socket server. JSON (JavaScript Object Notation) is a lightweight text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for most languages. JSON is widely used in this project, because both web socket protocol and web worker protocol are worked with using messages. Data exchange between main browser thread (web page) and web socket thread is done in the same fashion: objects are serialized and passed as strings.

wsClient also can accept objects from server and execute different methods depending on messageType. Every object passed in JSON format in degas framework has field messageType, which describes what that object is. For example, message types for server messages are stored in degas.consts.serverMessage and can be initial data, updated data, leader borad and global stats.

2.1.4 Web Worker interface

Web worker that is responsible for most of the computation is stored in 'evolve.js' file. Since web worker is executed in separate thread, it is completely separated from global scope and does not have access to DOM. Messages are used to communicate with web worker.

As with client-server interactions, messaging system is used to pass objects in JSON format. Once again, messageType filed of passing/receiving object tells what type of object it is. These types are described in degas.consts.workerMessage:

- **Log**. Web worker does not have access to widely used in web browsers console.log() function allowing to output any data in debug console (firebug in firefox or inspector in google chrome). This message type is a replacement.

- **Error**. This message type is used to describe an error occurred while executing web worker.

- **UI Update.** This message type is used to describe an update for user interface components on the web page. For example, if we want to show generation counter and current sum of fitnesses on the web page where web worker is issued then this message type is used to pass new information from worker to host page.

- **Best individual**. This message type is used to describe an object that stores the best individual sequence and its fitness. This object is to be send to the server.

Now the structure of the client looks like this:

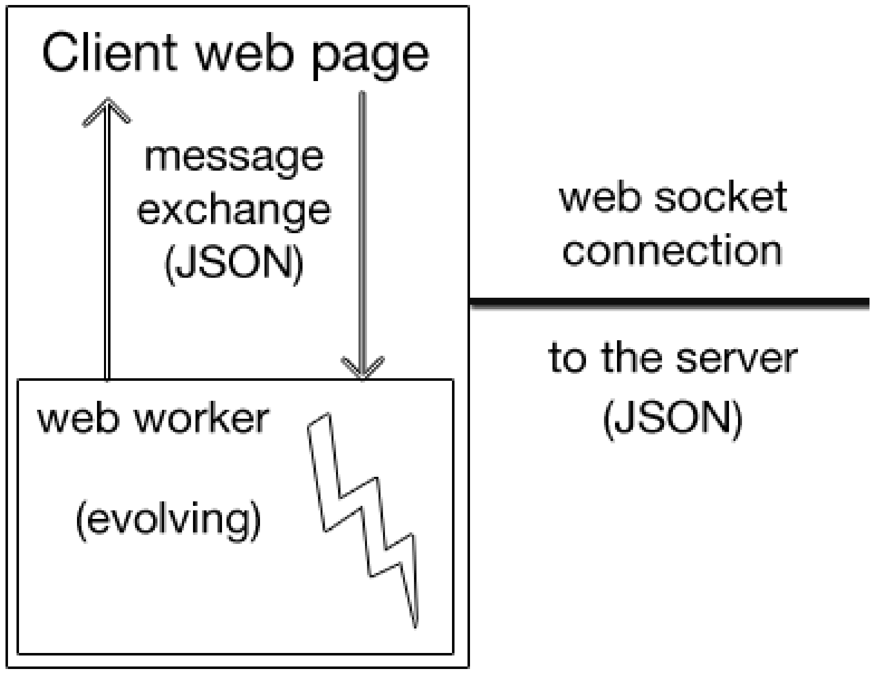
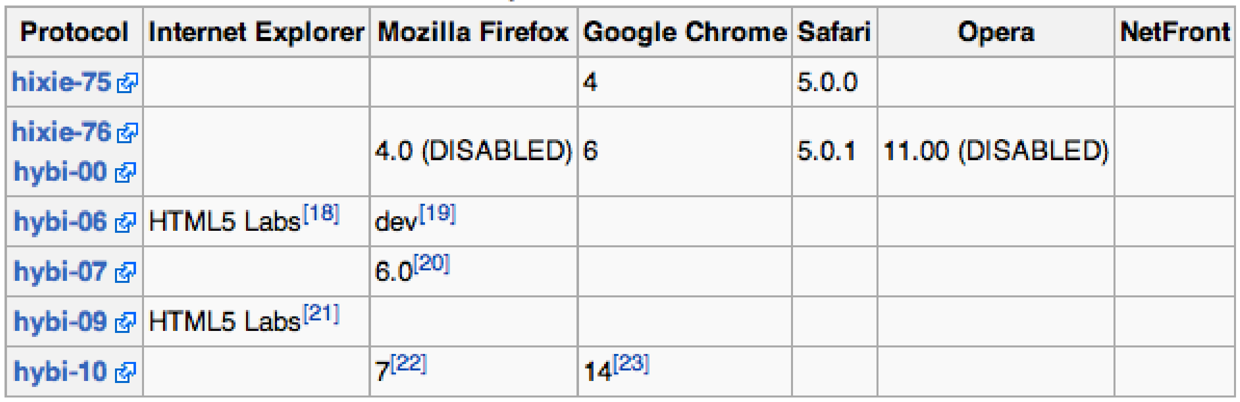


Figure 6. Structure of the client.

2.2 Server

The best way to bring javascript to server side is to use node.js. Node.js is an event-driven I/O server-side JavaScript environment based on V8. It is intended for writing scalable network programs such as web servers. The Google V8 JavaScript Engine is an open source JavaScript engine developed by Google and shipping with the Google Chrome browser.

To bring web socket functionality to node.js WebSocket-Node module was used. It supports the latest working draft of web socket protocol draft-ietf-hybi-thewebsocketprotocol-10. Today only Google Chrome v. 14 and Mozilla Firefox v. 7 support the latest web socket draft protocol.



Protocol	Internet Explorer	Mozilla Firefox	Google Chrome	Safari	Opera	NetFront
hixie-75			4	5.0.0		
hixie-76 hybi-00		4.0 (DISABLED)	6	5.0.1	11.00 (DISABLED)	
hybi-06	HTML5 Labs[18]	dev[19]				
hybi-07		6.0[20]				
hybi-09	HTML5 Labs[21]					
hybi-10		7[22]	14[23]			

Figure 7. Implementation status of web socket among web browsers.

The web socket server is not entirely backwards-compatible, so it will not necessarily accept web socket connections from older browsers using the previous versions of draft. There is not much we can do now, except for wait until the protocol is completely standardized.

Globally, about 42% of web browsers currently browsing internet support one or another version of web socket protocol.

The way node.js starts a web socket server is by starting a HTTP-server on port 8080 and attaching a web socket to it. This means a client must connect to ws://host:8080.

After starting server awaits for initial message “client-connected”. It is the only message that is sent as raw string, while all other messages are serialized into JSON format. When “client-connected” message is received server stores generates unique client id, attaches it to the connection and send initial data to the client.

Initial data by default is empty object. User can provide any object to be sent as initial data.

Other message web socket server can accept is “best-individual”. Once again, messages are stores in degas.consts.clientMessage. When server receives best individual from some node it stores it in cliTable: a hash table with connection id as a key and received object as value.

3.0 Deployment

Let us try to deploy a default dummy distributed genetic algorithm using degas framework. The server side requires no modification and is ready to be used. The only thing required is node.js installed and web socket module connected. These instructions are suitable for any Unix operation system (Linux, Solaris, BSD, Mac):

```
git clone --depth 1 git://github.com/joyent/node.git
cd node
git checkout v0.4.11 # optional.  Note that master is
unstable.
export JOBS=2 # optional, sets number of parallel commands.
mkdir ~/local
./configure --prefix=$HOME/local/node
make
make install
echo 'export PATH=$HOME/local/node/bin:$PATH' >> ~/.profile
echo 'export
NODE_PATH=$HOME/local/node:$HOME/local/node/lib/node_module
s' >> ~/.profile
source ~/.profile
```

Alternatively, for Windows there are pre-built binaries available at http://nodejs.org.

After that we need to either manually install WebSocket-Node module (by downloading it from https://github.com/Worlize/WebSocket-Node and putting the directory to ~/node_modules folder) or by using Node Package Manager (NPM):

```
# curl http://npmjs.org/install.sh | sh
# npm install WebSocket-Node
```

Now we need to download the latest version:

```
git clone --depth 1 git@github.com:freetonik/degas.git
```

Now we can use wsServer.js from degas framework to start web socket server:

```
node wsServer.js
Sun Aug 21 2011 16:33:43 GMT-0400 (EDT) Server is
listening on port 8080
```

This indicates that the server is up and running and ready for clients.

Now to setting up the client. We need to create a simple HTML-page and include main javascript file degas.js:

```
<html>
<body>
<script src='degas.js'>
</body>
</html>
```

This will add all necessary class definitions, methods and variables into global namespace. Now we need to create a web socket client instance to be able to talk to server:

```
<html>
<body>
<script src='degas.js' />
<script>
    var client = new wsClient(serverAddress, "8080");
</script>
</body>
</html>
```

where serverAddress is IP-address of web socket server. For local server (running on the same machine as clients) it can be set to “127.0.0.1” or “localhost”. At this point once the webpage is loaded the client web browser attempts connection to web socket server. When it connects and sends initial “client-connected” message browser console should show “Client connected” log message, and server log should state:

```
Client connected (client address=clientAddress, id=ID)
```

where clientAddress is IP-address of client machine and ID is unique number.

Now let us set up a web worker and define some message exchange infrastructure. A slightly better version of this code comes with degas.js so no need to type it in (it stored in test.html):

```
<html>
<body>
<script src=’degas.js’ />
<script>
    var client = new wsClient(serverAddress, ‘8080’);
    var worker = new Worker('evolve.js');
    worker.addEventListener('message',
function(message) {
    var receivedObject = JSON.parse(message.data);
    switch (receivedObject.messageType) {
        case (degas.consts.workerMessage['LOG']):
            console.log(message.logMessage);
            break;
        case (degas.consts.workerMessage['ERROR']):
            degas.errorHandler(receivedObject.data);
            break;
        case (degas.consts.workerMessage['UI_UPDATE']):
            updateUI(receivedObject);
            break;
        case
(degas.consts.workerMessage['BEST_INDIVIDUAL']):
```

```
            sendCandidateToServer(receivedObject);
            break;
        default:
            degas.errorHandler("unknown message
received from webworker");
    }

}, false);
</script>
<p id="latest_result"></p>
</body>
</html>
```

As you can see we also have added a paragraph of text with id=”latest_result” onto HTML page. We should also provide two functions: one for updating UI (in our case that paragraph of text) and one for sending new best individual to the server:

```
function updateUI(receivedObject){
    document.getElementById('latest_result').textContent =
receivedObject.generation + ":" + receivedObject.fitness;
};

function sendCandidateToServer(receivedObject){
    receivedObject.messageType =
degas.consts.clientMessage['BEST_INDIVIDUAL'];
    client.send(receivedObject);
};
```

Now, when client receives initial data from server via web socket it starts the execution of web worker. Worker begins evolution and each generation sends an object with messageType “update-ui” which changes the text in paragraph to display current generation and fitness sum (fitness of population). Every 10 (by default, value stored in degas.config.serverUpdateCycle) generations client sends best individual sequence and its fitness to the server.

Of course, there is no much use in this state: evolution tries to evolve individuals with highest bit sum and eventually will generate a sequence with all ones (no zeros). Yet, this generic structure works and now it is time to extend the framework to solve some real problems.

3.1 Extending the framework

3.1.1 Server

In this section we will try to extend the degas.js framework to solve classical longest common subsequence problem (LCS). The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences (often just two). Note that subsequence is different from a substring.

Searching for the longest common sequence (LCS) of multiple biosequences is one of the most fundamental tasks in bioinformatics. Biological sequence can be represented as a sequence of symbols. For instance, a protein is a sequence of 20 different letters (amino acids), and DNA sequences can be represented as sequences of four letters A, C, G and T corresponding to the four sub-molecules forming DNA. When a new biosequence is found, we want to know what other sequences it is most similar to. Sequence comparison [3-5] has been used successfully to establish the link between cancer-causing genes and a gene evolved in normal growth and development. One way of detecting the similarity of two or more sequences is to find their LCS.

We will have DNA molecules represented by letters A, C, G and T. Luckily, two bits is exactly what is needed to represent four letters, so let us decode them as following:

A – 00
C – 01
G – 10
T – 11

So, the first thing we need to do is to define decode function in the global namespace. Recall, when this function is not present degas will just decode every cell to sum of bits and return it as character. Our decode function must take array of sequences as parameter and return single array of characters. Both arrays must have the same length, but the input array is actually an array of arrays (cells), when output array is an array of characters.

```
function decode(sequence){
     var outputSeq = new Array(sequence.length);
     for (var j=0; j < sequence.length; j++){
          if (sequence[j] === [0, 0]) outputSeq[j] = 'A';
          else if (sequence[j] === [0, 1]) outputSeq[j] = 'C';
          else if (sequence[j] === [1, 0]) outputSeq[j] = 'G';
          else if (sequence[j] === [1, 1]) outputSeq[j] = 'T';
     }
return outputSeq;
```

You can imagine writing the inverse function that converts characters into bits:

```
function encode(sequence){
     var outputSeq = new Array(sequence.length);
     for (var j=0; j < sequence.length; j++){
          if (sequence[j] === 'A') outputSeq[j] = [0, 0];
          else if (sequence[j] === 'C') outputSeq[j] = [0, 1];
          else if (sequence[j] === 'G') outputSeq[j] = [1, 0];
          else if (sequence[j] === 'T') outputSeq[j] = [1, 1];
     }
return outputSeq;
```

Now we can store DNA sequences in any format and convert between them. Let's assume the DNA sequences are stored on the server in binary form and, for sake of simplicity, all client nodes are going to work on the same set of sequences. Of course, in real world it would make much more sense to divide the long sequences into smaller chunks and let each node work on its separate set. Then combine the resulting substrings to get global solution.

Degas.js server side has function getDataToSend() which returns an empty object by default. This function is called every time new client node is connected and you probably assume this is the place where problem data is divided into smaller parts. If so, your assumption is correct and even though for this simple example we will not divide data, it is easy to think of the way to do so here. getDataToSend function returns empty object only if prepareData function is not present in server's global namespace. So, let us define that function. It must return an object with initial data:

```
function prepareData(){
     var obj = {};
     obj.sequence1 = [[0,0], [0,0], [1,0], [1,1], …… [0,1], [1,1]];
     obj.sequence2 = [[1,0], [0,0], [0,0], [0,1], …… [1,1], [0,1]];
     return obj;
}
```

Of course, storing data in the function itself is not a good idea. That's why degas.js provides 'fs' variable to access file system. Please, refer to node.js documentation on file access. To keep this example simple we will stick with data stored statically in the function. Note, that accessing file system is one of the slowest operations server can do. So, loading data from file at start and storing it in program namespace as arrays is probably a good idea after all.

At this point, assuming the client is defined as well, server is practically ready: it can accept new connections, send initial data, accept best individuals and store them in the hash table of clients.

It is time to move on to the client.

3.1.2 Client

The most important function is computeFitness. It evaluates the bit string (that represents a potential solution) and returns numeric value of its fitness or quality. Recall, by default degas.js returns the sum of bits of all cells, which is not particularly useful. For longest common subsequence problem we want to evaluate a sequence by comparing it to two sequences sent by server as initial data. Once again, to keep everything simple in this example, we assume that the solution we're looking for has the same length as input sequences.

computeFitness function will return bit sum only if 'fitness' function is not present in the global namespace. We need to have access to the initial data (two sequences) sent by the server. They are stored in degas.serverData. If that value is null then no data was received by the client.

Let us write the fitness function, which takes the sequence and number of bits per cell as parameter and returns fitness value as number:

```
function fitness(sequence, bitsPerCell){
      var matchMatrix = new Array(sequence.length);
      var sequenceWalker = 0;
      var fitnessToReturn = 0;
      for (var i = 0; I < sequence.length; i++) {
                  matchMatrix [i]=0;
      }

      for (var i=0; i< sequence.length; i++){
            for (var j= sequenceWalker; j< sequence.length; j++){
                  if (degas.serverData.seq1[j] == sequence[i]) {
                        matchMatrix [i]++;
                        sequenceWalker = j+1;
                        break;
                  }
            }
      }

      sequenceWalker = 0;
      for (var i=0; i< sequence.length; i++){
            for (var j= sequenceWalker; j< sequence.length; j++){
                  if (degas.serverData.seq2[j] == sequence[i]) {
                        matchMatrix [i]++;
                        sequenceWalker = j+1;
                        break;
                  }
            }
      }

      for (var i = 0; i < CANDIDATE_LENGTH; i++) {
                  if (matchMatrix [i] === 1) fitnessToReturn += 1;
            if (matchMatrix [i] === 2) fitnessToReturn += 2;
      }

return fitnessToReturn;
};
```

This code snippet defines a match matrix and then iterates over two sequences looking for matches. Each match increments a value in the match matrix by one and if some character in given sequence is present in both DNA sequences then it is a part of longest common subsequence. The function then iterates over the match matrix and sums up its values into a single fitness value.

A good idea would be to use some variables instead of hard coded "awards" for a match (1 and 2 used in this example). This way we can map those variables to some UI component on the web page and allow user dynamically change awards for matches along with other evolution-related values (population size, etc). It is important to provide such functionality, because evolutionary computations are pretty unpredictable, and good results can be achieved by picking the "right" (or we should say lucky) combination of parameters.

So now we have everything ready for solving longest common subsequence problem in somewhat distributed fashion using evolutionary computation.

Roadmap

The goal for the next version of degas framework is to implement the following features:

- Allow variable sequence length and cell size, so that different individuals within the same population can have different sizes.
- Allow individual exchange between nodes via server. This is easy to implement within current version since we already have tools to send and receive individual sequences.
- Allow client nodes to connect to each other directly for data exchange.
- Add more complex crossover and mutation functions.
- Provide HTML templates with UI components mapped to genetic algorithm settings

References

1. A fast parallel algorithm for finding the longest common sequence of multiple biosequences [Yixin Chen, Andrew Wan, and Wei Liu, BMC Bioinformatics. 2006; 7(Suppl 4): S4]
2. Asynchronous distributed genetic algorithms with JavaScript and JSON [Merelo-Guervos, J.J.; Castillo, P.A.; Laredo, J.L.J.; Mora Garcia, A.; Prieto, A.; Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence)]
3. Berkeley Open Infrastructure for Network Computing http://boinc.berkeley.edu/
4. BOINC http://boinc.berkeley.edu/, University of California.
5. Computational Biology notes by Pat Morin http://cg.scs.carleton.ca/~morin/teaching/compbio
6. Computational Molecular Biology course notes [Evangelos Kranakis , Carleton University, School of computer science, 2010]
7. Computational molecular biology. Sources and methods for sequence analysis http://www.abebooks.com/Computational-Molecular-Biology-Sources-Methods-Sequence/3170439215/bd
8. Cyberaide JavaScript: A JavaScript Commodity Grid Kit [Gregor von Laszewski, Fugang Wang, Andrew Younge, Xi He, Zhenhua Guo, MarlonPierce, Grid Computing Environments Workshop, 2008. GCE '08]
9. Evolutionary computation course notes(genetic algorithms, genetic programming), complex adaptive systems. [Franz Oppacher, Carleton University, School of computer science, 2009-2010]
10. Generic workers: towards unified distributed and parallel JavaScript programming model [Adam Welc , Richard L. Hudson, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, 2010, PSI EtA '10 Programming Support Innovations for Emerging Distributed Applications]
11. Handbook of Evolutionary Computation [Thomas Back, David B. Fogel, Zbigniew Michalewicz [IOP Publishing Ltd. Bristol, UK, UK ©1997]

12.JSDC http://jsdc.appspot.com/
13.MapRejuice https://github.com/ryanmcgrath/maprejuice
14.Node.js http://nodejs.org/
15.Perl for Bioinformatics and Internet http://bip.weizmann.ac.il/courselprog/
16.Ravan http://www.andlabs.org/tools/ravan.html
17.SETI@HOME http://setiathome.berkeley.edu/
18.Web Sockets http://dev.w3.org/html5/websockets/
19.Web Workers https://developer.mozilla.org/En/
20.Wikipedia http://en.wikipedia.org