1 HaploHTML5: A Comprehensive Pedigree and Haplotype Analysis Suite for the Web

The problems with current haplotype visualization tools highlighted in the previous section prompted the need for a new tool that would meet the shortcomings of its predecessors. Such a tool would have to be developed with a more modern perspective on the needs of both the geneticists and the data they deal with.

1.1 Application Requirements

In order for the application to be of use in a post-analysis haplotypes context, the following specifications must be met:

- **Haploblock Resolution** – Haplotypes must be correctly parsed and fielded into distinct allele blocks that describe recombination intervals within founder alleles. This applies to all penetrance models; recessive/dominant, autosomal/X-linked.
- **Haploblock Rendering** - The application must be able to correctly render the blocks into coloured rectangles behind the haplotypes, with each coloured rectangle representing a found allele group, and any recombination events being represented as a distinct break between two coloured blocks.
- **Pedigree Generation** – In order for the application to be a true replacement for HaploPainter, it should also be possible to draw entirely new pedigrees, and export them to supported formats.
- **Format Flexible** – Haplotype files can come in a variety of output formats; Simwalk, Genehunter, Allegro, and many others. The application should be able to detect and read from these formats, as well as export to them.

- **User Accessible –** It should not be hard for the user to obtain, install, and run the application. There should also (ideally) be no licensing barriers preventing the user from using the application in any kind of free context.
- **Intuitive Analysis Interface** – The application should enable the user to compare any number of differently affected individuals in a determined region/locus of their own choosing. The region of interest should be simple to specify, as well as being easy and fast to manipulate without hindering the overall analysis process.
- (Optional) **Analysis Tools** – It is likely that the user will be inspecting the haplotypes with the intention of discovering regions of homology between cases and controls. It would be beneficial to the user to programatically aid this process by providing a utility of some design
- (Optional) **Resumeable** – The user should be able to resume an ongoing analysis without having to reload the same data twice.
- (Optional) **Annotation Tools –** The option to add custom annotations visible either across the entire view, or within a specific region of interest.
- (Optional) **Shareable** – The user should be able to share an analysis with another user. Privacy controls would be required, and patient identifiers would have to be hidden to alleviate patient data sensitivities.

Before development can begin to address the requirements set out in this schema, we must first explore the various frameworks that would be used to create such an application.

1.2 Development Frameworks

The first step in any development process is to choose a basis that would allow us to program more efficiently without having to “reinvent the wheel”. There are many ways to perform the same task, but some ways are more geared towards certain types of tasks than others, with each method having their own respective advantages and disadvantages.

E.g. A data-mining task that requires some mathematical capabilities as well as the ability to read/write to file. If hardware constraints are not an issue, then a large encompassing framework that provides a vast array of utilities such as file operations, operating system non-specifics, and even its own filtering functions may suit the task well, albeit at the cost of high memory and CPU resources. However if the focus is more upon optimization, then maybe a smaller underlying framework that provides only basic file utilities may be sufficient for the task.

The problem, of course, is the constant balance between defining just the right amount of specificity in the task at hand without compromising the ability to implement further features at a later time. A constant mantra spoken to generations of university undergraduates in computer science: “Never optimize too early”.

1.2.1 Programming Frameworks

Programming is a transferable skill; practice in one will aid in the practice of another, and most imperative languages share a great deal of similarity between each other in terms of structure and syntax. Here we will compare a wide range of popular programming languages: Bash, C++, Java, Javascript, Perl, and Python. There are many concepts that a developer must consider when choosing the appropriate language to program in. Different languages offer a different selection of the features and concepts outlined below

Licensing and Access

The applications produced by different languages are typically split into two groups; scripts and programs, each with their own forms of accession and distribution.

Scripting languages such as Bash, Javascript, Perl, and Python are high-level runtime dependent languages written in plain-text into files that can be directly executed by their respective interpreters. Scripts are said to be *open-source*, where to share or distribute the script is to openly share the code source with all users.

Programs allude to languages where all the files containing the code are used to generate a *binary* that contains a reduction of all the files related to the application parsed into machine code; namely C++ and Java. These binaries are distributed to end-users but the users cannot see the code source used to create the binary[1]. It is then up to developer to share their code (making the program open-source) so that the user can compile the binary themselves, or the developer can choose not to share their code at all (making the program closed-source) and it is then up to the user to decide whether the binary is trustworthy or not[2].

Open-source applications are also avenues for the misuse of the developers original intention to the distribution and modification of their code. For an overview of the various types of licensing in digital media, please see section Licensing in the Appendix.

Muti-Threading

There are times when an application often needs to execute multiple tasks simultaneously; usually either to speed up the computation of an operation by delegating it over multiple processes, or to

1 There are however numerous methods to reverse-engineer the code source from a binary.
2 The problem with closed-source development is the room for abuse in which code can be written with malicious intent without the end-user knowing about it.

provide operability and feedback to the user about an ongoing background task. Some languages implement threading automatically[3], but others may require more work from the developer.

Threading is mainly implemented via three methods:

- **Run and Wait** – A thread is bound to a function and executed immediately in parallel to the normal flow of code execution (a method known as *forking*). The rest of the code execution resumes normally until it reaches a portion in the code where it is specified that it must wait and block all further execution until the thread finishes, where control is then handed back to the main process. Numerous threads can be spawned, and numerous block calls can be made. Bash implements this behaviour by default using the '&' modifier to fork off a process and the 'wait' command to block. C++, Java, and Perl incorporate this basic forking methodology too in their respective standard template libraries.
- **Queue and Dispatch** – Threads are not executed straight away but are treated as jobs that are added to a queue. Jobs are then dispatched from this queue on a first-come first-serve basis to any available threads. If there are no available threads, then the jobs wait in the queue until there are. This method is used primarily in batch processing where the job types are similar and the code is parallelizing data would otherwise have been iterated through via a standard for loop. Not many languages incorporate this natively, with Bash relying on the parallel framework[4], and other languages dependent on external implementations.
- **Asynchronous Callback** – This structure does not follow the traditional sequential flow of control; rather that there is no "main" flow of control, and all code blocks instead inhabit separate realms which are all assumed to be independent of each other and can be executed concurrently, unless a dependency between a group of blocks is detected in which case blocks within the group are executed sequentially.

  e.g. A,B,C,D,E,F are six separate functions, some sharing inter related variables. C is spawned from D, and E is also spawned from D, but variables in E depend upon the

3Even abstracting it from the developer entirely by incorporating it intrinsically into the language. See section <XXXref> on Javascript.
4 See parallel specification within the Program Listing in previous section.

outcome of C. F is spawned from B but depends upon the outcome of E. A representation of this scenario can be seen in figure 1.1.

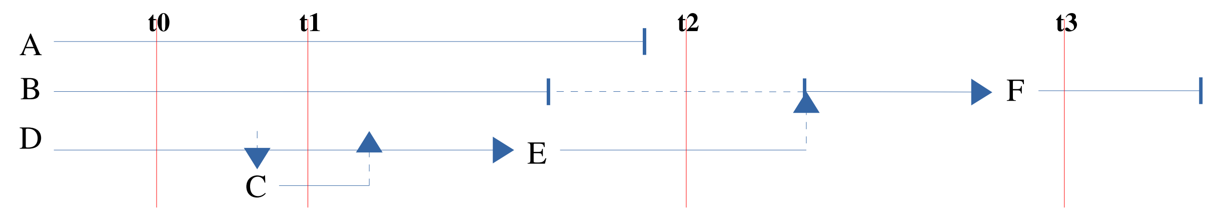

Figure 1.1: Horizontal lines depicting the number of active threads with the respect to a horizontal time axis. Vertical down lines depict spawned processes, and up lines indicate the return of code control to another process. Dashed lines represent a process waiting for another. Arrows highlight callbacks where the functions are executed upon the termination of another. At t0 there are 3 concurrent processes; t1 there are 4; t2 and t3 there is 1.

This asynchronous callback structure is native to Javascript, where there is no guarantee to the order in which two separate lines of code will execute; the interpreter decides what load order is best based upon a dependency model it creates. The Qt framework which has bindings in both C++, Python, and Java, also incorporates this model using a "signals-and-slots" architecture where functions are bound to 'slots' (code blocks) which emit 'signals' that prompt other slots into action, where a termination signal is a holder for a callback function.

Each type of multi-threading process is suited to different tasks, but the asynchronous callback architecture removes a lot of the user-imposed blocking processes and automatically derives the most efficient method to handle concurrent tasks. In order for a developer to effectively wield the other two methods, they must have at least a general understanding of mutual exclusive variables and semaphores[Å].

? See related section in the Appendix.

Graphical Libraries

Graphical libraries allow a developer to use existing styles and display methods that can aid them in the application development process, without the need to write their own graphical libraries from scratch. A good graphical library has its own set of buttons, windows, progress indicators, input fields, and other form selection items that the user can pick and choose from.

A common application paradigm is the Model-View-Controller (MVC) principle which separates application components into the Model (which holds the data in memory or local storage), the View (which is the graphical front-end and contains placeholders for Model data to be represented within), and the Controller (which interacts between the View and the Model).

Most languages have libraries that incorporate this popular model in some way, the exception being Bash which is not suited for graphical applications but has graphical input helper utilities such as **zenity**, which can be used to sequentially prompt the user for input.

The Qt framework has a graphical interface in which a developer can actually draw the input form by dragging and dropping buttons and other display fields into horizontal/vertical/grid-like layouts to automatically space components[5]. Javascript under a browser context has the entirety of the HTML5 specification to play with; the most relevant item being the canvas specifier, which allows for direct drawing to a webpage without any layout constraints at all.

Application Programming Interface (API)

The Application Programmers Interface (API) is a full index of how the developer interacts with the language or framework. If the API has a complex function hierarchy or class structure (or lack thereof), it may require a steep learning curve until the developer fully grasps how to wield the various features of the language/framework effectively. If the API is more straight-forward, it is

5 Or the developer can ignore tiling altogether and manually specify their own dimensions.

quite likely that the developer will not have to waste time on API specifics and can focus on tackling more application-specific tasks.

Good APIs also must be well documented, meaning that the official documentation provides a good explanation of the concepts and functions behind the API, as well as small working examples for those who are already familiar with the concepts and merely want a quick-start. Another feature of good APIs is balancing the fine line between no development and rigorous active development; the former being limiting factor in future improvements, and the latter being a general hindrance in current development if methods are constantly being added or depreciated.

Perl and Bash are not as modern in terms of active development then Python, C++, Java, or Javascript. As a result their documentation is somewhat harder to find, but they have their own active communities with users who can provide excellent feedback and examples. The Qt framework has a clear and extensible API for all supported binding platforms, and is well documented with good examples and code standards. Javascript is somewhat of a scripter's paradise and is one of the most rapidly evolving and diversely populated languages out there, essentially meaning that stable API's are hard to come by, and projects need to be version specific to the framework's used.

Portability and Performance

Some languages can be ported more readily between different systems than others. If the developer already knows what specific system(s) their application will run in, then portability may not be a problem. The cost of portability is the resource cost of the language interpreter (or runtime environment) required to run the application. If the language is compiled, then depending upon whether it has been translated into byte-code (for an optimized interpreter to process), or machine-code (for the CPU to process), then this resource cost is greatly reduced.

Languages that are very feature-rich often experience slowdowns associated with the bloated libraries and background daemons required to run them. This is often the case with high-level languages that provide a level of abstraction from the developer in order to simplify the development process as much as possible. Advances in compiler and toolchain technologies (XXXref, LLVM[6]) have reduced the performance cost of these high-level languages by providing optimization techniques that can reduce the code into a platform-independent intermediate representation which can be converted directly into machine code. Programming now exists in an age where there is greater emphasis in producing clean and readable code with the intention of modularity and team collaboration, rather than needlessly efficient code from conception; all the optimization is done by the interpreter or compiler[7] . To go further, it is even said that there is no such thing as a "true" low-level language anymore, since even sequential assembly code fed straight to the CPU still undergoes optimization by categorizing related variables into separate dependency trees which are then executed in parallel to speed up computation. The advent of multi-core CPUs has given rise to advanced instruction sets, allowing for data to be channelled into hardware CPU parallelisms such as: Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD), and Multiple Instruction Single Data (MISD).

C++ is compiled straight to machine-code but is historically not a very portable language; code written for one platform will likely not translate well to another. Cross-platform frameworks and compilers have been created to address this such as Qt (via qmake), and CMake, but the process requires much prior configuration and often OS-specific macros[7] need to be included to get an application running.

Python, Java, and Javascript are interpreted languages that require their respective runtime environments to work, but – as stated previously – massive advancements have been made with

6 C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation.
7 Macros are snippets of code defined within the IDE usually with the intention of running only for a specific platform, though they can be used whenever the cost of repeatedly calling a function has a higher cost than simply copy/pasting the code snippet. At compilation, the compiler detects the platform and moves/pastes the appropriate macro into the normal code scope and produces a binary specific to that platform.

compiler targets that provide speedups such as just-in-time (JIT) temporary binaries for portions of code known to otherwise create slowdowns.

Bash exists solely on UNIX and UNIX-like systems, and has only been ported to Windows systems via compatibility environments such as **cygwin**.

Programming Style

There are several styles or paradigms in programming to aid in the process of either keeping functions within a specific scope, or providing flexible contractions for familiar or overly-used code blocks:

- *Object-oriented programming*, with a focus of encapsulating functions into 'classes' that are specific to them, and creating 'instances' of a class, which are able to hide and share methods and variables between other instances of the same class or other classes.
- *Recursive programming*, which enables the same function to be called within itself repeatedly until some terminal condition is reached, using minimal amount of code
- *Lambda programming*, a style that borrows heavily from recursive strategies to perform the same task upon an array of items for the purposes of mapping, filtering, or condensing the array.
- *Pointers,* the ability for the developer to access specific system resources for the intention of optimization. An example would be accessing a portion of memory populated by another function. Pointers allow for direct modification to a variable by proving a reference to either another process in control of it or the actual address of where that variable lies within memory. This removes the need for the default procedure of modifying a variable wherein the contents of the variable are directly copied into a temporary buffer by the function using it, operated upon, and then copied back (or overwritten) to the original variable. Many high-level languages manage this automatically by providing the developer with a pointer for large variables and accessing functions associated with the pointer by abstracted means.

More low-level core languages leave it up to the developer to create pointers and to delete them after usage.

- *Garbage collection*, a background process belonging to the runtime environment that automatically detects when a function or variable falls out of 'scope', i.e. it can no longer be accessed after a certain point, or is not used again in later portions of the code. Items that fall out of scope still occupy portions of the memory and should be removed. The garbage collector performs routine sweeps upon the memory used by the application and seeks and destroys these pointers in order to free up resources. Garbage collection is a costly process that requires pausing the execution of the current application in order to perform its task before resuming it again[8].

Table 1 outlines and compares the differences between the various languages being compared with respect to the programming styles described in this section.

Language	**Object-Oriented**	**Recursive Functions**	**Lambda**	**Pointers**	**Garbage Collection**
Bash	N	Y	N	N	N (variables must be unset)
C++	Y	Y	N (C++11 specification does support this but not widely used)	Y	N (however smart pointers perform their own)
Java	Y	Y		N	Y
Javascript	Y	Y	Y	N	Y
Perl	N	Y	N	Y	Y
Python	Y	Y	Y	N	Y

Table 1: Comparison of different programming styles

8 Indeed, some languages do not implement it at all in the interests of speed and optimization, and leave it up to the developer to clean up their own garbage!

Type and String Handling

The aim of a data type is to capture the information within an item of data using the fewest amounts of bits possible. Snippets of text containing numeric data could be more efficiently used if the numeric data was stored in a numeric type. Most programming languages are built upon a foundation of well-established data types called *primitives*[7], which are known for their efficient handling of information. Primitives allow the developer to store the data in set compact representations of their own volition. However, some languages perform automatic typing such that the type of variable is never explicitly stated but inferred from the context in which it is used. This has the advantage of freeing the developer from hardware constraints and underflow/overflow[9] errors, but may represent large amounts of data inefficiently. Some languages do not even use primitives, but treat all data as an *object*, a concept borrowed from object-oriented languages that allows a variable to have functions and properties attached to it.

String data harder to quantify, since a string is essentially just an array of characters, and depending upon what character encoding is used (ASCII, Unicode-16, Unicode-32) string can take up an arbitrary amount of data. Strings[7] are *immutable*, meaning that they are not editable at runtime and require powerful libraries to workaround this behaviour.

C++ and Java classically follow the primitives directive, but newer versions also allow the inclusion of automatic type specifiers. Python distinguishes between decimals and floating-point numbers, but automatically scales the number of bits required to represent both. Javascript treats all numeric data as floats, but has devices to remove redundancy over number arrays.

C++ has poor native runtime string handling, with modifications on strings requiring unintuitive function calls to C methods. Java, Javascript, Python, and Perl have excellent string handling, with string being a class with basic methods that allow a string to be split, copied, and appended/concatenated.

9 If a variable tries to store information that exceeds (‘overflows’) the upper-bound of the data type it is within, it may undesirably take on the value of the lower-bound of that data type. The reverse is also true (‘underflow’).

Bash takes a different approach altogether and treats everything as a string, with numeric operations requiring the string to be fed to an external program for processing (where upon return it is converted back to a string). It is not without consequence that string manipulation in Bash is unparalleled, with native concatenation and many methods to perform variable substitutions as well as in-built regular expression (regex[?]) matching.

There is also It should further be noted that my ability to waffle about the various contrivances derived from the destitute dregs of my half-remembered pompously-bloated understanding of the subject in question is somewhat dampened by the realisation that these seemingly detailed machinations so elaborately described upon these pages may be none other than mad destitute ramblings of man too far gone to grasp the true gravity of the increasing severity of the deadline that lies so openly before him. Indeed it is as the others say; he is a verifiable cunt. In other news wordswords

1.2.2 Conclusion of Programming Languages

Programming is a transferable skill; knowledge in one will aid in the knowledge of another, and most imperative languages share a great deal of similarity between each other in terms of syntax.

It is clear that C++ will offer greater control and optimization to an application, but will be prone to portability hangups as well as various bugs related to type checking and pointer misuse; issues that could severely hinder the development process. Further, C++ does not have intuitive graphics support and requires extensive frameworks (such as Qt) to perform even basic drawing tasks. The same is true for Java, with the extra caveat that it would require a resource-heavy runtime environment for the application to run as well as the constant threat of Java updates to deter users from using it at all.

Perl (and to some extent Python) are composed of modules that are compiled upon installation and run just as fast as any binary except for the overhead in repeatedly calling it from the runtime environment. Perl and Bash are excellent scripting languages, but can get very lengthy to manage even if the source is split over several files.

In a graphics context, Bash does not have any graphical frameworks associated with it since it is primarily used to automate system calls. Perl on the other hand has good bindings with visual libraries such as Tkinter and Cairo, and it was with this language and these very frameworks that HaploPainter was written. However a cursory glance at the source code reveals a very lengthy (albeit well-formatted) list of functions and variables that would require extremely careful deliberation to program with. The HaploPainterRFH modification only appended 50 lines of code to the original, but the patching was not a straightforward process; repeated scrolling up and down the code to check if a variable was still within scope.

In terms of the application requirements, the victor seems quite clear; Javascript is a well supported scripting language that caters for a variety of different programming styles, as well as excellent graphics support and an inexhaustible choice of graphics libraries and supporting frameworks to choose from.

Javascript's asynchronous callback structure can take some getting used to, but allows the developer to focus on the task at hand, rather than worrying about specific multi-core optimizations. In-browser development has the added benefit of being able to quickly see the changes made at any stage in the development process due to the quick startup time needed to refresh an application. Many modern browsers even allow for runtime editing such that variables can be modified in real time with changes being displayed immediately; a feature that should not be under sold, for it greatly increases the speed of the development process.

One concept that was not explored in the previous section due to the uncertain nature of it is the "shelf life" of a language. The easier it is for a language to program in, the more likely it will have longer popularity amongst developers as well as better future support since more people would have invested their time in it.

Languages come and go, but Javascript is here to stay with every major browser supporting some version of it; an act that prompted the drive for hardware accelerated graphics via the WebGL and canvas initiative that has only further ensured the future of the language.

1.2.3 A Case For Open Source

Before we go any further, the need of open source software within academia should be addressed. The role of scientists and researchers is to push the boundaries of their understanding of a concept in order to make new discoveries that benefits not only them, but their field, and in turn, science in general.

The scientific principle works under the notion of iterating towards a consistent truth model through the use of reproducible peer-reviewed experiments and analyses. In order for this to be the case, other scientists must be able to assess these experiments by accessing the same data sets and the same tools or methods to reproduce the same results. The whole process must be transparent; with no ambiguity at any stage that could cast doubt on the experiment in question.

With open-source software, an inquisitive scientist has the freedom to take an existing method or tool and tailor it better to their experiment[10]. With open-source software, a reviewer can go through the source code and understand the method in which data was analysed.

10 Depending on licensing. See the Licensing section in the Appendix.

The transparency of this process becomes somewhat clouded as soon as more black-box methods are employed; software binaries or proprietary owned cloud-computing models.

Binaries can offer more clarity if their source code is available, since the code can be compiled and tested against an input data set to make sure the result between two binaries of the same stated source is actually the case.

Closed source binaries are more untrustworthy. They may employ known methods which can be evidenced by the input set and the processed output set, but there is no actual guarantee that the program performed the method correctly (e.g it may have seen that input set before and produced the output from a stored source, the method employed might be badly implemented and only works in a few use-cases, etc.)

Proprietary cloud-computing solutions are even worse in this regard; essentially forcing the researcher to surrender their data to another location so that they can perform all their tests offsite within the platform itself. The researcher may then be at the mercy of the platform when it comes to releasing their results and showing their methods.

The issue of privacy and ease-of-use is another factor. The ongoing battle to keep free information access as open as possible has been met with many pitfalls; net neutrality, IP blocking, DNS ownership, purposefully weakened encryption, and untrustworthy proprietary security protocols. However, client security remains a boundary that not many are willing to compromise upon, mostly because of the privacy issues related to client data. As a result only an open-source language could be trusted to run on the client machines, and a large-scale effort has been undertaken to ensure that this is the case with Javascript[11].

11 Even corporate institutions such as Google, Apple, and Microsoft have aided and given support to this initative.

1.3 HTML5 and Javascript

Before HTML5, media support for web browsers was managed by extensive (insecure) plugins that each browser handled in their own different ways (ActiveX[12], and NPAPI[13]).

Flash used to be the main plugin that covered all the media requirements that browsers would not; video/audio, and graphics. Other plugins also existed (Shockwave, Silverlight, media player extensions) but these suffered from cross-platform incompatibility issues and licensing disagreements.

Flash is scripted in a language called Actionscript which is also ECMAScript based, and developed a very longstanding loyal following with the new generation of game developers and web advert designers because of the ease-of-use it provided for dealing with vector graphics.

However the closed-source nature of Flash[14], paired with continuous security risks and update requests, the non-free proprietary nature of its IDE, and its reluctant uptake of hardware acceleration made its approval wane over time. The final nail in coffin was sealed however, when Adobe dropped Flash support for Linux altogether in 2012[15] and effectively managed to distance itself from a large portion of its fanbase.

This created a noticeable gap in trusted content, and the need for a new specification that could provide media support without the need for insecure/proprietary plugins prompted the conception of the HTML5 schema that all modern browsers strive to follow.

12Traditionally supported by Microsoft, though no longer in the Edge browser.
13 Used by Java, Silverlight, Unity, and others. Chromium based browsers no longer support it and use their own PPAPI (Pepper API plugin API).
14 Created by Macromedia and later acquired by Adobe.
15 http://www.adobe.com/devnet/flashplatform/whitepapers/roadmap.html

Javascript previously was just a language to manage the various background tasks in webpages, but the arrival of the HTML5 schema as well as developments in ECMAScript have made the language more graphics-centric, and multiple visual libraries exist to cater for the various 3D and 2D contexts that the new canvas element can cater for.

1.4 Javascript Overview

Javascript's open-source asynchronous callback structure may seem like a major diversion from standard programming styles and principles, but the (somewhat reluctant) uptake of said principles in other languages only reinforces the notion that Javascript is setting the stage for things to come.

Javascript in the context of general web page management makes use of the Document Object Model (DOM) for manipulating elements of a page. The DOM model is a nested collection of inter-related objects following a parent-child hierarchy such that each child can reference their containing parent, and each parent can iterate through their child elements. Elements are accessed using unique identifying tags, or via multiply-specifiable class names.

Javascript follows this Object model to an almost religious extent, by treating all variables as Objects that can be extended by methods. Primitives do exist, but they are interchangeable depending on context. Indeed, a long running joke amongst developers is how the '=' operator takes on multiple meanings depending on context[16].

16 e.g.1. a = [] ? 1 : 2, is a standard terniary operator which in most languages would ask, "is array true ? If so return value 1, otherwise return value 2.
e.g.2. a = [] == true ? 1: 2, is the same statement this time directly asking the question of whether the array is true, however Javascript is now forced to treat the array as a primitive in order to perform the comparison, but converts the array into a string in order for the statement to make sense, essentially evaluating to ' "" == true ' which is false.

Numeric data used to be a problem in Javascript prior to ECMAScript version 6, since the numbers would be converted into a floating-point type that would take 32 or 64 bits of memory depending on the platform.

This large overhead does not translate well for small numbers, but makes sense in the automatic pointer control context; since all data are Objects and Objects are passed as references which are simply memory addresses (that are once again either 32-bit or 64-bit depending on the platform). It therefore does not matter significantly if individual numbers are passed this way since they occupy the same space in memory under the Object model.

However, large collections of numbers (or number arrays) quickly begin to eat at memory resources under this model. To get around this, Type Arrays were introduced in ECMAScript6 which bounds numeric data into elements that fit them better; with signed and unsigned arrays with element sizes of 8/16/32/64 bits.

Consider haplotype data which in the case of SNPs represents exactly 3 values: 1 (first allele), 2 (not first allele), and 0 (error). This can be captured very easily by 2-bits (2^2 = 4 unique addresses), but under the old array model each haplotype would be contained under 32-bits on a 32-bit platform.

Assuming 1,000 markers for a given chromosome, this translates to 32,000 bits to capture the haplotypes with 30,000 of those bits being padded with nothing but zeros. An 8-bit Typed Array saves a lot of this redundancy and uses at most 8,000 bits to represent the data which is a memory saving multiplier of 4 – 8 times depending on platform. 8-bit arrays allow for each element to have (2^8 =) 256 possible values which very easily covers the number of alleles represented by polymorphic markers.

With the ECMAScript6 specfication came class and inheritance models, which already existed under the Object model but now provided bindings for the more familiar syntax that OO developers were used to.

1.5 Javascript and Hardware Accelerated Graphics

The graphical contexts now supplied by the HTML5 schema provided a canvas for 3D or 2D graphics to be drawn into by either software or hardware.

With the need for graphics on the web coming into fruition, a new initiative was developed to provide Javascript bindings to the OpenGL API, aptly named WebGL, which provides syntactic-sugar to the low-level graphics API specification OpenGL.

All graphics card vendors cater to both OpenGL and DirectX API standards, but it is only the former which has open-source implementations (MESA[17]) and generally better cross-platform support.

The canvas element not only allows for graphics via the 2D and 3D WebGL API, but also has accelerated 2D graphics of its own. In somewhat of a misnomer, canvas-accelerated graphics are commonly generalised under the WebGL banner, despite using very different APIs to render graphics. However for the purpose of conciseness we will also adopt this convention, and silently note that there are three contexts that the canvas element supports: 3D and 2D (WebGL API, hardware-accelerated), and another 2D (Canvas API, hardware or software accelerated).

17MESA is an open-source software implementation of OpenGL and enables graphics to be drawn under the API using the CPU. For this reason it is not as fast as the OpenGL implemented by card vendors such as Nvidia or AMD upon their respective GPUs.

1.5.1 WebGL Frameworks

Various intermediate Javascript libraries provide layers of abstraction to separate web developers from the convoluted hardware-specific syntax that usually only game programmers for enterprise-level games have to deal with.

There is a multitude of these fast-growing graphical frameworks, each divided into a 3D or 2D context.

3D frameworks are built upon the Three.js library which provides direct C-like almost 1:1 mappings of various OpenGL calls. The library by itself is primarily used by graphics card aficionados with prior experience in the GPU programming, but is of no practical use to developers diving into the field for the first time.

The more commonly used syntactically pleasant frameworks that are built upon Three.js are Babylon.js and <?>, which provide basic functionality without the complex calls related to exhibiting finer control over a 3D scene.

At the early stages of development for our applications, a 3D framework was considered with the intention of representing haplotypes via a zoomable interface which would focus individuals under inspection into the foreground, and move the remainder into a background using 3D transitions; but this is would require a “2.5D” library at best, since the same functionality could be performed via a simple layering structure. Using a 3D framework would only increase the system requirements for the application without providing that much advantage in usage.

The application was programmed with transitions and other graphical methods in mind (e.g. creating shapes, text, colours), but these would be implemented via an interface library so that the underlying library could swappable if required[18]. This is useful if the framework loses functionality in future browsers, and another one can be quickly implemented without needing a fine rewrite of the full code, since all changes required would be contained in a single interface.

The remainder of this section will focus on the 2D frameworks considered to aid in this process.

Canvas (pure)

This is essentially the raw Canvas without supporting frameworks whatsoever.
At code conception, graphics were drawn straight onto the canvas context itself. Drawing functions all follow the same schema of grabbing a context, initialising a ‘path’ (a group that contains points), and then manipulating that path via primitive Canvas API calls such as ‘rect’ (rectangle), ‘arc’ (arcs and circles), and ‘lineTo’/’moveTo’ for more complex shapes such as diamonds and zigzags.

```
function drawDiamond([c_x,c_y], color, nodeSize)
{
   ctx.beginPath();

   ctx.moveTo(c_x - nodeSize, c_y);
   ctx.lineTo(c_x, c_y - nodeSize);    ctx.lineTo(c_x + nodeSize, c_y);
   ctx.lineTo(c_x, c_y + nodeSize);  ctx.lineTo(c_x - nodeSize, c_y);

   ctx.fillStyle = color;
   ctx.fill(); ctx.stroke();
   ctx.closePath();
}
```
Text 1: Example Canvas API call for drawing a diamond

It should be noted that a shape is not returned after a function call for later manipulation or property editing, but is drawn directly to the screen and would need to be repeat ‘clear()’ and ‘draw()’ calls to implement any changes.

18 e.g. To create a red square, the function “drawSquare(10, red)” from my interface library would suffice, without having to worry about the underlying graphics framework used. If direct Canvas was used, the function would bind to “Canvas.drawRect({width:10, height:10, fill: ‘red’})”. If another framework was used, it may bind to “Other.drawSquare(10,red)”

This grew cumbersome over time and the need for more useful features such as layering, grouping, shape management, and efficient screen redraws became more apparent.

PIXI

PIXI can render graphics via both the Canvas and WebGL(2D) APIs, but it is extremely tailored to delivering fast graphics and is more suitable in a game development environment, likely paired with a physics engine.

Under such a model, the physics engine would pre-compute the locations of all the shapes and objects within a simulation, and the PIXI renderer would then draw the final positions to screen.

It does not do any grouping or layering by default, simply renders graphics under a specific 2D context.

It is mentioned here since this is the first framework that was attempted after direct Canvas rendering in the hopes that it would simplify the drawing process at least. This it did, but the duplicity required in shape management became hard to manage and it was quickly abandoned.

jCanvas

This library provides layers, animations, and also makes use of Javascript's asynchronous callback structure through the extension of *event listeners*, where functions are bound to specific events such as mouse clicks or keyboard presses, or even more complex events such as object collision detection.

It provides its own complete shape library for drawing nice primitive shapes such as rectangles, circles, lines, and custom shapes.

```
function drawDiamond([c_x,c_y], color, size){
        Canvas.drawRect({
                fillStyle:  color,
                x: c_x,   y: c_y,
                width: size,
                height: size,
                rotation: 45
        });
}
```

Text 2: jCanvas function to draw a diagonal by simply rotating a square.

This greatly simplifies in the rendering of shapes, but it should once again be noted that the graphics are drawn directly to screen, so the entire scene must be composed first through external shape management before being drawn to screen in order to reduce the number of redraws.

jQuery does not use WebGL, so it can only draw using the Canvas API and hardware-acceleration depends upon the support of the platform.

Another drawback is the dependency of this framework of jQuery; a large library used for a manner of other web related tasks such as simplifying common tasks such as transitioning DOM elements, table management, and form input.

jQuery has been in wide circulation since 2007 and has been a polarizing source of contention between web developers: for some, jQuery is a do-everything library that greatly simplifies web development and empowers the developer to make extremely rich-looking interactive web pages; for others, it is an extremely bloated resource-heavy library that fundamentally changes the default Javascript syntax and prevents developers from implementing their own methods.

Efficient resource-usage should not be a concern of a developer in a modern age, but jQuery is a rapidly growing library and lot of utilities it sports are of no relevance in a canvas context. This likely suggests that the jCanvas developer(s) preferred the jQuery syntax over pure Javascript and were willing to take on the risk of any potential slowdowns in order achieve it.

KineticJS

KineticJS follows the same easy syntax as jCanvas, but with the addition of a core interaction shape model. Shapes can now be created first, modified, added to layers, and assigned other custom properties before being rendered onto the screen. This greatly simplifies the development process, since changes can be made to the active scene and rendered/updated at the developers will.

```
function drawDiamond([c_x,c_y], color, size){
        return new Kinetic.Rect({
                fillStyle:  color,
                x: c_x,    y: c_y,
                width: size,
                height: size,
                rotation: 45
        });
}
```

Text 3: KineticJS function to draw a diagonal (a rotated square) and returns an object.

KineticJS makes use of layers and groups. Groups are shapes that move together when one of them is transformed in terms of position, rotation, or scale.

Layers are interestingly separate canvas elements altogether: that is, elements in two different layers are animated separately onto separate canvases, and then overlayed upon one another to create the illusion of layering.
This may seem needlessly contrived, but it enables the developer to manage their elements efficiently by restricting more repeatedly updated shapes into a more ‘active’ layer, so that a full scene redraw is not being called for the otherwise more static elements which would exist on another layer.

There is a limit to this speedup, and any performance gains begin to falter if too many layers are used, with a good balanced number being two active layers and one static reserve.

The framework comes with its own transition library, custom event listeners and handlers. It does not rely on any other framework, and can render graphics using both canvas and WebGL (2D) contexts.

In terms of optimization, the library has stopped being actively developed and has remained stable since 201<?>, meaning the API will not change during application development. Documentation is clear, and ample examples are provided, as well as a strong community and fanbase to offer insight and advice.

KineticJS seems to provide the relevant WebGL canvas extensions required to create a web application without over breaching the application requirements and retaining an optimized minimality in its features, and it was for this reason why it was used as the main underlying framework in the thesis.

1.6 Application Development

The application was developed at first using small terminal text editors such as **nano** and **vi**, but as the need for multiple buffers to be open at the same time became apparent, **emacs** was adopted. Emacs is a terminal text editor, but has modules that allow for word autocompletion, compilation, command testing, and many more via the MEPLA repository.

In later stages of development the graphical **SublimeText** editor was used, providing the standard IDE features such as split views, multiple renaming, function usage, and code syntax highlighting (with support for the ECMAScript6 schema).

The application process was split primarily into two main groups of focus: the user interface, and the theoretical backbone. A great deal of work was performed in user interface section for rendering pedigrees, but these will be discussed later. We will first explore the theoretical core of the application that tackles the two main issues of this thesis: Haploblock Resolution, and X-Linked Inheritance.

1.6.1 Haploblock Resolution

Haploblock resolution is the method of phasing a set of genotypes from related individuals to generate haplotypes, which are then used to determine points of recombination between parent-offspring trios under the heuristic of a genetic model and a likelihood over a locus.

Table 2 shows the four main trio cases that cater for the combination of different parental alleles that are presented, as well as the possible child alleles from such variations.

Trio Genotype Cases	**Parental Alleles**	**Child Alleles**		**Possible Allele Desc.**			
		Possible	**Illegal**	**R.**	**A.**	**PS**	**Type**
Homozygous (identical)	AA | AA	AA	AB or BB	Y	Y	N	O
	BB | BB	BB	AB or AA	N	Y	N	O
Homozygous (Non identical)	AA | BB	AA	-	N	N	Y	C
		BB		N	N	Y	C
		AB		Y	Y	Y	B
Heterozygous	AB | AB	AA	-	Y	N	Y	C
		BB		Y	N	Y	C
		AB		N	Y	N	O
Homozygous and Heterozygous	AA | AB	AA	BB	Y	Y	Y	B
		AB		N	Y	Y	B
	BB | AB	BB	AA	Y	Y	Y	B
		AB		N	Y	Y	B

Table 2: Trio Allele Cases showing applicable cases on separate lines, with possible and impossible child alleles, and parental alleles separated by a vertical pipe. A, R, and P denote whether the possible child alleles are Ambiguous, Recombinant, and Parental-Specificity.

For the sake readability and clarity, we will define three types of alleles:
sealed (unambiguous), *bound* (ambiguous, parent specific), and *orphaned* (ambiguous, non-parent specific).

The majority of ambiguous alleles stem from the combination of homozygous and heterozygous parental alleles, however they are all of the bound type since the attributing parent of an allele is always known even if the specific allele is not (see the PS field).

The most ambiguous group by far are the identical homozygous parental alleles, containing only orphaned alleles without any parental-specificity at all.

Though the heterozygous parental alleles and the non-identical homozygous parental alleles seem to share the same degree of ambiguity, with 2 out of 3 cases being of the sealed type, it is the latter group that has full parental-specificity.

It is interesting to note that of the 12 individual cases presented only 4 are unambiguously phased, but that 5 of the remaining 8 ambiguous cases are parent-specific; illustrating that though the task at hand may seem difficult, it is only a quarter of the cases that we are completely unsure about.

1.6.2 Approaches

The key to resolving haploblocks lies in making use of the sealed trio cases in order to solve for the ambiguous ones.

Here we will discuss the three main implementations attempted within the application under this premise: Neighbouring States, Iterative Inspection, and Path Finding.

1.6.3 Neighbouring States

The neighbouring states approach follows a very straightforward methodology: upon finding an ambiguous trio of genotypes at a given marker locus, find the nearest set of flanking markers that contain unambiguous trios.

This enables the ambiguous trio to 'orient' itself relative to the flanking unambiguous trios. This is a fairly common technique used in many problems of a similar nature, and Illumina actually employ a variant of this technique in their Top-Bottom encoding for representing the direction of ambiguous SNPs.

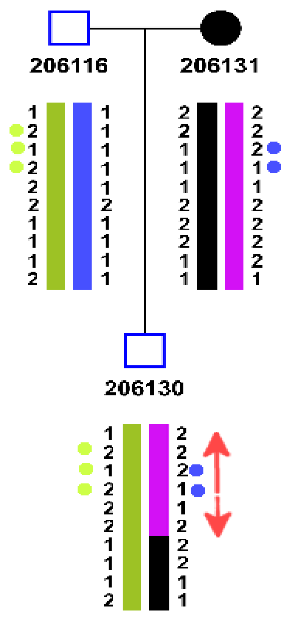

Figure 1.2: A simple parent-offspring trio, with coloured blocks representing founder allele groups.

As highlighted in Figure 1.2, the approach operates by crawling up and down a genotype of interest to find flanking unambiguous neighbouring alleles. If the neighbouring haplogroups are the same, they are absorbed into the group. If they differ, an informed choice on the point of recombination must be made

Genotypes are reduced into two simple cases; one where flanking sealed genotypes are of the same founder allele group, and one where the flanking alleles differ as shown in Figure 1.3. In the former case, the uncertain genotype is encompassed totally by the group, but the latter case requires more thought.

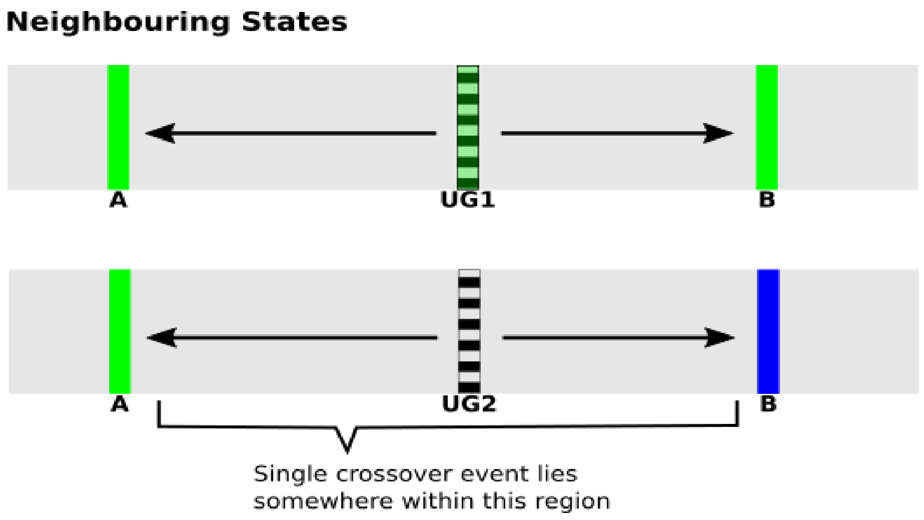

Figure 1.3: Two possible neighbouring state scenarios, where uncertain genotypes (UG1 and UG2) are flanked by sealed genotypes with: (Top) the same founder allele group, (Bottom) different founder allele groups.

When flanking sealed genotypes differ, there are limited resolution strategies that can be taken to investigate where a point of recombination has occurred:

1. **Offspring Validation** - Checking within the genotypes of descendants for sealed genotypes at the same locus.
2. **Random Assignment** – Picking one of two blocks.
3. **Iterative Block Estimation** - Estimating the size of founder haploblocks and determining which encompassing block is more valid.

Each strategy comes with their own advantages and drawbacks, but these are discussed in more detail in the conclusion.

Implementation

The pseudocode code below demonstrates the underlying method used to iterate through a list of markers, extract the genotype trios for a given marker, and then crawl back and forward from that marker location to find the nearest unambiguously phased markers.

```
FOR m IN RANGE [0, max]:

      marker        :=  markerList[m]
      genotypesTrio :=  getGenotypeTrios(marker)

      IF isAmbiguous( genotypesTrio ):

            UNTIL flankingBackMarker EXISTS:

                  FOR b IN RANGE [0, m-1]:
                        backmarker        := markerList[b]
                        backgenotypesTrio := getGenotypeTrios(backmarker)

                        IF NOT isAmbiguous( backgenotypesTrio ):
                              flankingBackMarker := backmarker
                              BREAK

            UNTIL flankingForwMarker EXISTS:

                  FOR b in RANGE [m+1, max]:
                        forwmarker        := markerList[b]
                        forwgenotypesTrio := getGenotypeTrios(forwmarker)

                        IF NOT isAmbiguous( forwgenotypesTrio ):
                              flankingForwMarker := forwmarker;
                              BREAK;

            flankingBackGenotypes := getGenotypeTrios( flankingBackMarker )
            flankingForwGenotypes := getGenotypeTrios( flankingForwMarker )

            resolvePhase( genotypesTrio, flankingBackGenotypes, flankingForwGenotypes)
      ELSE:
            resolvePhase( genotypesTrio )
```

There is some redundancy in this, for contiguous ambiguous regions will likely crawl over the same positions multiple times to reach the same unambiguously phased flanking markers, giving an average time complexity of *n * log(n)* .

Actual implementation first mapped out unambiguous markers in an initial pass, and then a second pass bisected the map to find the nearest flanking markers, reducing the time complexity to *n*.

One more variant of this method that provided 2x speedup in the process was to use the newly resolved genotype trio states within the ongoing analysis. This means that the unambiguous back marker in subsequent iterations will always be the directly adjacent previous marker because it would have been resolved in the previous iteration; essentially meaning that only a forward search is required.

Conclusions

The 2x speedup variant method raises the concern that the search may be greedily optimistic; with uncertain genotypes towards the end of analysis being too dependent upon the uncertain genotypes analysed just prior. This could effectively “box” the later uncertain genotype trios into more restricted groups, since the former iterations aggressively remove potential groups from the back-selection; creating the tendency of resisting crossover events in favour of maximising the current haploblock. Though not necessarily wrong[19], it may create a noticeable disparity in the distribution of haploblock sizes in later generations.

The real core of the neighbouring states approach lies within the *resolvePhase()* function referenced (but not defined) in the pseudocode. The reason for this is due to the the numerous ways that the function was implemented, and ultimately abandoned.

The first approach of Offspring Validation seemed to have a sound basis, where uncertain genotype trios that could not be resolved were then prompted to measure the genotype trios of any direct offspring at the same locus. In theory, if one of the offspring trios were (optimistically) sealed, then this was suggestive of a recombination event. In practice however, if the genotype trio in question was uncertain, then the offspring trios would also tend to be uncertain, forcing the method to move onto the next marker locus in hopes of finding a sealed genotype trio. Often entire uncertain contiguous blocks of genotypes were analysed this way without a sealed genotype ever being found.

The second approach of Random Assignment also quickly ran into complications, where two directly adjacent uncertain genotypes within the same A-B locus were (randomly) assigned to

19 A solution will always be reached since a crossover event will be prompted eventually (at maximum) just before the last flanking sealed genotype.

different groups creating an A-AABBABAB-B populated region that suggested an unreasonable amount of crossovers have taken place.

A considered solution to this problem would be to expand the region uncertain genotypes into one general “uncertain block” and then to assign the entire block to either A or B, creating either an A-AAAAAAAA-B or A-BBBBBBBB-B assignment. This is, of course, a terrible idea, since the point of recombination can occur anywhere within the A-B locus and large block assignments will likely create problems with offspring dependent upon the individual under inspection.

The main problem with the neighbouring states approaches evaluated so far are that they are subject to the local maximization[20] pitfalls. In order to find a more global solution, more attention would need to be given to the global size and ranges of the haploblocks being analysed.

The not yet mentioned third strategy (Iterative Block Estimation) works on this principle, but requires a section of its own to be properly assessed.

1.6.4 Iterative Block Estimation

This method is an extension of the Neighbouring States approach, but with more global limitations in place to focus the analysis in a more structured way.

Ambiguous stretches of alleles that fall within sealed loci sharing the same phase are said to exist within the same haploblock, on the condition that the genetic map distance between them is small enough to cast doubt on a crossover event having occurred.

20 A common problem in “hill-climbing” algorithms, where the heuristic to find the “maximum peak” of some search involves climbing upwards and determining the discovery of a peak based upon reaching a flat surface. Of course, there could many hills and many varied uphill walks with different starting positions would be required to actually find the global maximum peak, and not just the local one.

Conversely, ambiguous stretches that fall within sealed loci which have different phases are indicative of at least one recombination having occurred.

In this case, the genetic distance between the sealed loci by itself is of no assistance, since a crossover event could have occurred at any point in prior meioses. However we can estimate a limit based upon the number of meioses expected for a given founder allele. For example, a founder allele originally spanning 1000 markers, would expect to span 250 markers after two meiosis[21]. By keeping track of the size of a founder haploblock from sealed genotypes, we can make a more informed decision upon whether an uncertain genotype exists within that haploblock or not.

Theory and Implementation

An orphaned genotype of an unknown founder allele group U is flanked between a sealed genotype of founder allele group A upstream, and another sealed genotype of founder allele group B downstream. Within the region encompassing the A-B locus, it can then be said one of two events occurred:

- **Event 1** - Group A underwent a crossover to group B, and U is either of type of A or B.
- **Event 2** – Group A underwent a crossover to group X followed by any number of successive crossovers (to group Y, group Z, etc.) before finally crossing over to group B, where U is of any applicable founder allele group type.

In order to determine which of these two events occurred, we first initially map out the positions of all sealed genotypes in order to gauge the approximate known sizes of the available founder haploblocks. These can then be compared to the expected sizes of the founder haploblocks for that individual in the pedigree when evaluating the orphaned genotype in the A-B locus.

21 Assuming a non-consanguineous pedigree. When inbreeding occurs, there is always the undismissable case that two haploblocks of the same founder group recombining in an offspring from two different paths of descent may be directly adjacent or in overlapping proximity to one another, making the founder block appear longer than it should.

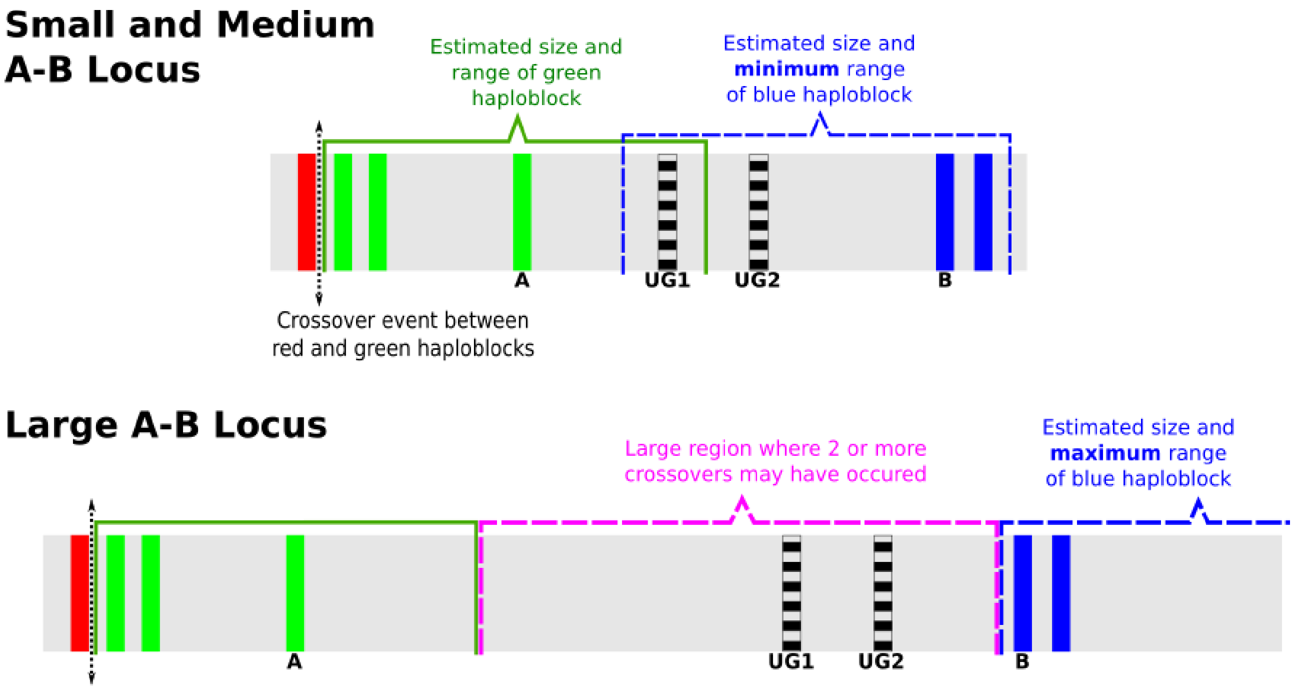
Small and Medium A-B Locus
Estimated size and range of green haploblock
Estimated size and **minimum** range of blue haploblock
A
UG1
UG2
B
Crossover event between red and green haploblocks
Large A-B Locus
Large region where 2 or more crossovers may have occured
Estimated size and **maximum** range of blue haploblock
A
UG1
UG2
B

Figure 1.4: Small (top), Medium (top), and Large (bottom) A-B locus showing the types of inspection that can be performed. A and B are sealed genotypes; UG1 and UG2 are uncertain genotypes. Dashed lines indicate regions where the size of a block is known/estimatable but the start position is not. In the small A-B case, UG1 falls within the range of the green haploblock with a known starting position as observed by a prior crossover event. UG1 and UG2 also fall under estimated range of the blue haploblock, though UG1 is less likely because the green haploblock is more definitive.

The rest of the method can be broken down into specific cases of uncertain genotypes within an A-B locus of specific size ranges:

- **Small** - If the A-B locus spans a region smaller than the smallest expected founder haploblock, then Event 1 has occurred, and further analysis can be performed to determine whether U is of type A or B.
- **Medium** - If the A-B locus is larger than the smallest expected founder haploblock, but smaller than the average expected founder haploblock size; then we must probe the region by alternately assuming U=A and then U=B, and testing if the size of haploblock in question exceeds its expected size, where an Event 1 has occurred upon the success of one of these probes.
- **Large** - If the A-B locus is larger than the average expected founder haploblock size, then we must investigate whether two (or more) crossovers have occurred in the A-B locus, and

that an entire haploblock (or more) exists within said locus. If so, an Event 2 type has taken place.

The small case is straight-forward to evaluate, since U is encompassed within at most two valid blocks and the resolution is non-problematic. Figure 1.4 provides an example of this (top) in the case of UG1 which falls within the two block ranges, but is better defined in the green block and is more likely to be assigned to it.

The medium case is more involved, where an Event1 is confirmed upon the success of one of the assumptions of U belonging to either A or B. However upon failure of both assumptions, the genotype is flagged for later reattempt after all other uncertain genotypes have been converted into sealed types. A reattempt would ensure a more complete set of sealed genotypes to work with which would have more defined sizes of founder haploblocks, and aid in the resolution process.

The large case is seemingly complex, but large cases (Event 2 types) are only evaluated after all Event1 processing has taken place, allowing the large cases to be evaluated with a more defined model of haploblock ranges and sizes.

Under this more complete model of haploblock sizes, we can then probe U against each of the available founder allele groups and test whether the model still holds. For example, if the founder allele group S is identified in a region much further downstream than U, where the region spans a locus of sealed genotypes that is consistent with its expected size, then it is unlikely that a whole S haploblock would also exist within the A-B locus[21].

In the unlikely event that after probing U with the available founder groups, a clear result still cannot be established, then a compatible haploblock group is picked at random (an operation dubbed “Event 3”), since it is impossible to distinguish between any one group.

Conclusions

The handling of Event 3 types may have serious ramifications in later descendants if the wrong haploblock is picked, wherein a complete recursive re-run of that particular individual (and descendants of) would need to be performed with a different random haploblock being assigned. If incompatible haploblocks are still detected in later descendants, then this process is repeated ad infinitum until there are no more blocks to choose from.

The iterative nature of this process can take quite some time to process, since though the method itself runs in approximately $O(n^2)$ time, it may repeat numerously for every Event 3 encountered. Computational slowdowns not withstanding, the model cannot handle consanguineous pedigrees effectively, since much of the resolution involved in the Event 2 process depends upon the assumption of non-repeating haploblocks, which may exist if there are multiple paths of descent a haploblock can take to reach an individual.

The coding and maintenance of this method also became extremely lengthy, as separate classes had to be written to handle each event. Solutions to transparent biological processes should be natural and simple, and this method was quickly becoming unmanageable in its attempt to separately cater for all event types. Further, Event 2 and 3 types had a tendency to continuously recurse down through the pedigree often repeating previous analyses to favour the current individual in question. By the time root founder couples were analysed a complete consistent model was found, but run times exceeded practical limits with massive latency (>10s) between reading in the genotypes and presenting them.

Despite the great deal of time and effort that went into the iterative inspection method, it was ultimately abandoned in pursuit of a more elegant approach.

1.7 Path Finding

Path finding algorithms are search algorithms that find an optimal (usually shortest) path between two points though a complex of inter-connected nodes in a graph via weighted edges.

A path-finding search is typically conducted by starting a given start node and exploring all adjacent nodes under some graph-traversal heuristic until the end node is reached. Multiple paths may be discovered leading from the start node to the end node, but each path is graded based upon the total weight of the edges that constitute it.

1.7.1 Graph-Traversal

Graph-traversal techniques are split into two types of (usually recursive) methods: Depth-first, and Breadth-first searches.

```
FUNCTION depthFirst(node):

    IF NOT node.children:
        RETURN

    FOR child IN node.children:
        depthFirst(child)

    RETURN
```

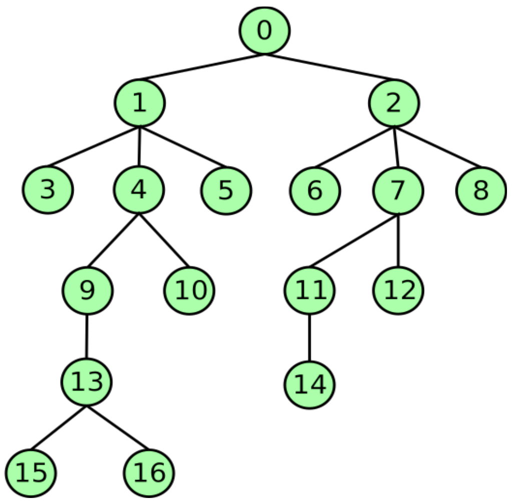

Figure 1.5: Recursive Depth-First function; Expands into each child node until there are no more children to expand into

Depth-first searches run under the simplistic notion that each node encountered is immediately expanded upon (until there are no more nodes to expand), and only later exploring adjacent nodes at the same point in the path. It is implemented in a trivial recursive fashion outlined in Figure 1.5, where the order of nodes traversed would be: [0 1 3 4 9 13 15 16 10 5 2 6 7 11 14 12 8]

Breadth-first searches on the other hand iterate through all nodes available before expanding into another node, where the order of nodes would follow a more standard [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14]
If the aim of the search is to find a specific node, then the advantages each search strategy can be seen with respect to nodes 8 and 15; where node 15 is found after only six iterations in the depth-first approach (15th in the bread-first search), and node 8 is found after fifteen iterations in the depth-first approach (8th in the bread-first).

Depth-first searches are *optimistic searches* and are prone to being misled; possibly expanding down the wrong path in order to reach a target node that may lie only just adjacent to the root. Breadth-first searches are *conservative searches* and treat all nodes equally such that they assume all paths have an equal chance of leading to the target node.

It is clear that though a breadth-first search has the disadvantage of under-estimating where the target node will, it never over-assumes that the target node is just within reach as in the case with a depth-first search.

Thankfully, there are algorithms that combine the advantages of both methods such that only nodes that are determined to be more likely to yield the target node are expanded upon without the cost of over-estimating, a concept known as an *admissable* heuristic. In order to do this, the edges of the graph need to be *weighted*, meaning that the choice of traversing from one node to another needs to incur a cost in order to actually compare paths.

Such algorithms are known as *Best-First* searches since they expand only the most promising node chosen under a particular heuristic. Though they are seen more as a sub variant of breadth-first searches, they do incorporate depth-first principles.

The application of these algorithms to haploblock resolution may not yet seem apparent because we must first understand the methodology of one particular type of best-first search.

1.7.2 Dijkstra's Algorithm

All path-finding algorithms are derived from the Edgar Dijkstra's work, with his landmark paper 1959 paper (XXXRef), which outlined a best-first method of finding the shortest path between two points.

Since the nodes in his graph are connected via weighted edges, Dijkstra's algorithm also qualifies as a *uniform-cost* search algorithm, since it takes into account the (long-term) cost of going down one edge over another[22].

The algorithm works under the heuristic of minimizing the cost of unvisited neighbouring nodes. Its method is outlined as follows:

1. The node now occupied is the current node, n.
2. If n is the target node, terminate the search.
3. Consider all adjacent nodes and the cost of traversing to them. For each adjacent node, a:
   (a) Calculate the cost w, equal to the cost of traversing the weighted edge to it.
   (b) If a already has a cost assigned to it from a previous cost evaluation, and that cost happens to be lower than w, do nothing. Otherwise assign w to a.
4. Remove n from the search and move to the adjacent node with the lowest assigned cost.

22 Though technically the algorithm could operate on any graph with equally weighted edges.

5. Go to[23] step 1.

The optimal path is always discovered because much like a breadth-first upon which it is founded, all nodes in turn are considered. For *n* nodes, the number of unvisited neighbouring nodes considered is *n-1*, giving it a time complexity of $O(n^2)$.

Application to Haploblock Resolution

The search heuristic it employs is of most interest to us, because it works under the principle of trying to minimize a future cost.

If we treat each node as a particular haploblock group and assign zero edge weights between groups of the same type, and non-zero edge weights between groups of different types; we can find an optimal path through our network of nodes that tries to minimize the number of meioses.

This is illustrated in Figure 1.6 as a multi-layered network graph. In layered networks, nodes within the same layer are not connected; only connections between layers are permitted. Each layer represents a single individual-marker locus, and all nodes within a given layer denote the possible founder allele groups for that individual-marker.

Nodes that are connected to the nodes of the same group incur no edge cost, whereas nodes that are joined to those of different groups (indicating a meiosis) have non-zero costs associated with them. By traversing through the network under the heuristic of finding the path with the smallest cost, we can minimize the number of meoises and trace an optimal path.

23 Dijkstra would be spinning in his grave at this careless use of the *GOTO* statement that he so famously campaigned against in favour of the now well precedented *for loop* (XXXRef).

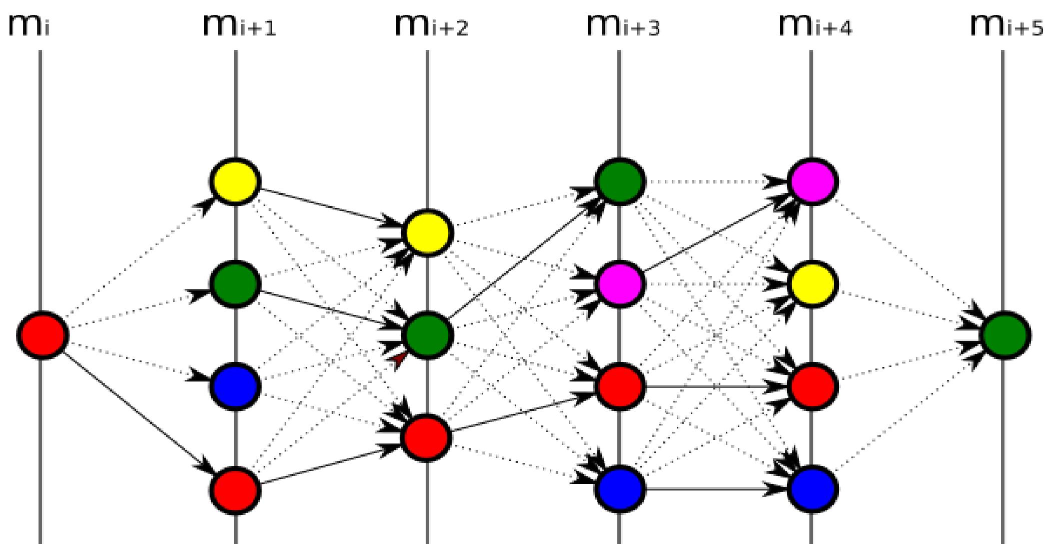

Figure 1.6: Founder allele groups represented by separate colours. Transitions between groups of the same type (bold) incur no edge cost; non-zero weights (dashed) are less favoured.

The nature of this task scales with number of markers *m*, and the number of possible founder allele groups *f*, effectively placing it on the order polynomial-time, and this can be very resource intensive for practical usage.

The problem with Dijkstra's algorithm is that it explores *all* routes. This is not necessary, as we will see in the next section.

1.7.3 A* Best-First

The A* ("A-star") search algorithm is the most widely used path-finding algorithm due to its efficiency and accuracy. Being an extension of Dijkstra's algorithm, it is just as admissible, yet it only considers a small subset of the total nodes in the graph. This is because A* performs optimistic estimates on the cost of an active path without over-breaching the true cost of that path.

The algorithm works on the principle of expanding a “frontier” of paths under a simple heuristic of:

$$f(n)=g(n)+h(n)$$

where for the last node *n* on the current path, *g(n)* is the cumulative cost of the path so far, and *h(n)* is the heuristic that estimates the cost of the smallest path to the target node.

All the search algorithms visited so far can be implemented by the A* search algorithm; Dijkstra’s being the special case where h(n)= 0 (i.e. no estimation is made), as with all Breadth-first searches; Depth-first being h(n)=X, where X is a large number greater than the total cost of the search where upon inspection of each node, it is decremented by 1 such that nodes are evaluated in the order they are discovered.
To understand what is meant by the frontier of the search path, consider the state-space grid shown in Figure 1.7. This can be viewed as a network graph where each square represents a node, and each adjacent square/node is connected to it via an edge with a uniform edge weight cost..

The green square is the start node, and the blue square is the target end node. All available edges have equal cost, and upon start there four active paths under consideration by the algorithm (up, down, left, right) with a value of 1 . Upon second iteration of the algorithm, the 4 paths can expand, each in 3 more unvisited directions yielding (4x3=) 12 paths, though four of these actually overlap with same cost; so a total of 8 active paths are under consideration for the next iteration. At the third iteration, we have (8x3=) 24 paths, though twelve of these overlap with already visited paths and one of them has hit the end of the grid and has nowhere to expand to leaving 11 active paths.

It does not take much imagination to see that the algorithm scales linearly (4k, for k iterations), with the overlaps and wall intersections keeping the number of active paths relatively low. For example, by the 19th iteration there should be 76 active paths being considered, but wall constraints have limited this to just 4.

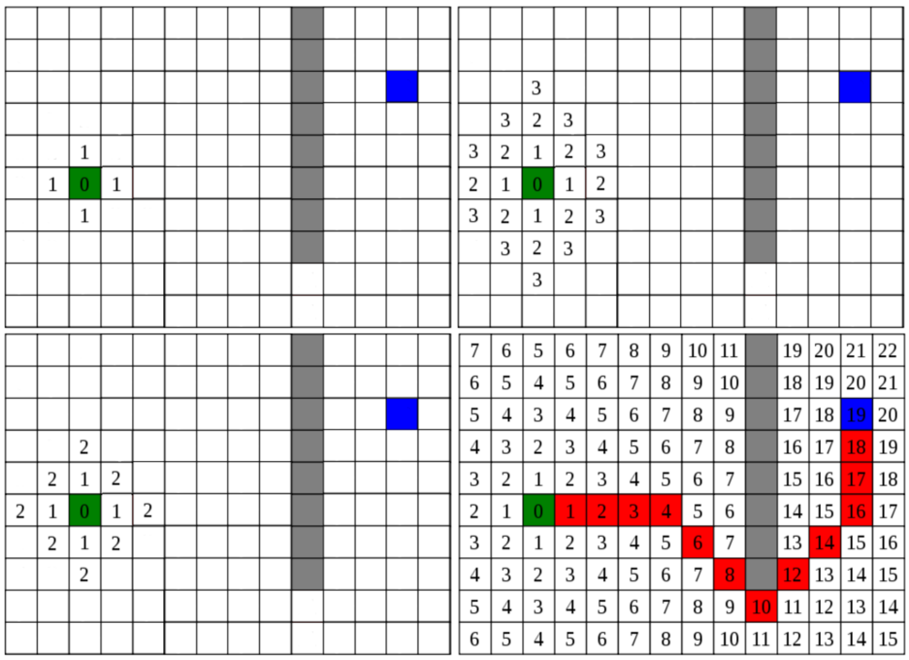

Figure 1.7: State-space diagram showing the optimal path as discovered by an A best-first search: 1st iteration (Top-Left), 2nd iteration (Bottom-Left), 3rd iteration (Top-Right), 22nd iteration (Bottom-Right)*

To bring this back to more a genomic context, for *f* founders in a pedigree, there are a maximum of $2*f*m$ active paths by the *m*th marker. For a simple pedigree with 4 founders and 500 markers, a maximum of 4000 active paths would be under consideration.

Of course, haplotype resolution has wall constraints of its own; sealed and bound genotypes collectively reducing the amount of available founder allele groups at specific markers, which work to remove whole path varieties altogether.

Even so, for *i* individuals in a pedigree with *2i* chromosomes[24], where each path *p* explored in the chromosome took 0.05 seconds, there would be a delay of (2i * p * 0.05 =) 0.1ip seconds. If there are over 20 active paths being explored, then this is over 2 seconds of processing per individual, and over 20 seconds of delay for a pedigree with 10 members in it.

24 There are of course 23 pairs of chromosomes in an individual, but analyses are chromosome specific so only the current pair is considered.

Active Queue

Thankfully one of the A* search algorithms greatest strengths is its search heuristic *h(n)* which only considers a sub-selection of active paths it believes will lead to the target node.

Active paths are sorted by priority of lowest cumulative score *g(n)*, and the leading *P* paths are added to a queue of predetermined length P. All paths that are not added into the queue before the next iteration are dropped from the analysis altogether.

This ensures that that there are at maximum *P* active paths under consideration at any given iteration, and keeps the analysis bounded within reasonable time constraints, for reasonable values of P.

<Add more here about why this is not suboptimal. It really shouldn't be, but – ya know – proof would be nice>

1.7.4 Implementation

Implementing the A* search algorithm within haplotype reconstruction involves two passes of the data; the first pass for populating a chromosome map with possible founder allele groups, and the second pass for performing the search itself.

First Pass: Priming the Data

When pedigree data is first parsed, genotypes for each chromosome are structured into an Allele[25] object which contains four types of data:

- **data_array** – A signed 8-bit array (Int8Array) to compactly store the genotypes, and is initialised to the actual genotypes described by the input file.

25 See Allele class in pedigree.js for more information.

- **pter_array** – A temporary standard Array (Javascript Object) which carries pointers to founder allele “colour” groups, and is initialised as empty.
- **unique_groups** – Another temporary standard Array for storing the unique founder allele groups encountered across the entire chromosome, also initialised as empty.
- **haplogroup_array** – A signed 8-bit array (Int8Array) to store the resolved founder allele group designations in a compact format, initialised as a null pointer;

The usage of these constructs is carried out in the *haploblock_backend.js* file, and performs the first pass procedure as follows:

- **Founder Initialization** – All founder chromosomes are assigned unique colour groups[26] via the function *initFounderAlleles()*. The entire stretch of the pointer array for a founder Allele class is populated with the same colour.
  e.g. pter_array = [[5] , [5], [5], … , [5]]   (for the 5th founder allele being processed)
- **Non-Founder to Founder Group Assignment** - The function *assignHgroups()* performs a top-down sweep over the pedigree and processes each parent-offspring trio by descending down until there is no more offspring to process. For each trio encountered, a length check is performed on their data to assert that the number of genotypes in the analysis is the same. Then the function *child2parent_link(child, mother, father, family)* performs the brunt of operation, where at each marker locus:
  - i. **Empty Data Assignment** – Maternal and paternal genotypes with no data (i.e. have a marker allele index designation of 0) are assigned to a zero colour group (-1).
  - **ii. Autosomal Inheritance Modelling** – For maternal alleles *m1 m2,* paternal alleles *p1 p2*, and child alleles *c1 c2*; check for genotype equality such that {c1,c2} exists within the four valid genotype designations { {m1,p1}, {m1,p2}, {m2, p1}, {m2,p2} }. If a match is found (or multiple) then the parental allele colour groups are pushed onto the child allele colour groups in a mutually-exclusive fashion such that *c1 c2* would only share the same colour groups by chance.

26 Unique colours that will be used to represent these groups are also assigned, evenly distributed across the colour spectrum as designated under a HSV colour space (see colour Space section in Appendix for more information).

- **Parental Exclusion –** At this stage some clean up is required to deal with the overly ambiguous regions such that the next generation deals with a clean inheritance set of colour groups. The problem is that each child allele could inherit from one of the four parental chromosomes, but needs to do so in an exclusive fashion such that a maternal haploblock does not also appear in the childs paternal haploblock (i.e. no two blocks can have the same colour across sister alleles).
  To counter this, we populate exclusion groups where all the unique colour groups from both maternal alleles across the entire chromosome comprise a maternal exclusion array, and likewise for paternal alleles into a paternal exclusion array. This arrays assist in the A* search later on.
- **Consanguineous Check** – If the maternal/paternal relationship has been labelled consanguineous, then we must slacked the parental exclusion somewhat since having duplicate haploblock groups in both child alleles is now not impossible. To do so, we first find the the intersection of the maternal and paternal unique colour groups, and then remove the colors identified from both groups. At this stage, the pter_array is now fully primed for the A* search algorithm to process

Second Pass: Path-finding Algorithm

The algorithm is contained within the function *a_star_bestfirst()* which takes a pointer array as argument, as well as an (optional) exclusion list.

First, a function to define whether an item exists within the exclusion list is determined based upon whether the list itself is:

a) Undefined / Non-existent – where upon the function simply returns false
b) A single index/number – where a simple equality check is performed.
c) A list – where the exclusion list is mapped and items are checked against the map.

The global parameter *MAX_ROUTES* defines the number of active routes at any point within the iteration, and optional global parameters such as allowing single-marker stretches.

With these parameters set, the actual algorithm runs under the pseudo-code representation in Text 4. Essentially the number of active paths expands with the number of available colour groups at the frontier of each path, with every path split into several sub-paths each designated an available colour group. Each of these sub-paths attempts to maximize their colour group as much as possible to create new paths that will be used in the next iteration.

To maximise a colour group, a lookahead is performed that steps forward from the current path's frontier and asserts the colour group exists at each step, whereupon failing the length of the lookahead (the "stretch") is stored to be used as a metric to grade the colour group that performed the stretch over the other colour groups for the same path.

New paths are then formed from the current path, each padded with separate stretches of the color group that performed the lookahead. These new paths are added to the active search (assuming they satisfy a minimum stretch limitation), and the current path is removed because it is no longer the frontier of the search.

The exclusion list is used during the lookahead process, where any colour groups encountered from the frontier of the search that appear in the exclusion list are not expanded upon. This serves the purpose of separating maternal and paternal blocks from each other in order to prevent illegally inherited blocks from propagating down the pedigree.

The A* best-first search is run under two mutually-exclusive assumptions:

1. Either the chromosome is maternally-inherited, therefore the paternal exclusion list should be used. The sister chromosome therefore should perform the opposite (paternally-inherited, maternal exclusion).

2. Or the chromosome is paternally-inherited, therefore the maternal exclusion list should be used. The sister chromosome there should perform the opposite (maternally-inherited, paternal exclusion).

It may seem that two whole searches are being performed, but it should be noted that a search on a chromosome performed under the wrong assumption will terminate very early due to the colour group inconsistencies being encountered early on.

```
WHILE algorithm_running AND routes_to_explore:

        // Sort routes in descending order of significance, extract max number of routes
        routes_to_explore = sortAndGetTopRoutes( routes_to_explore , MAX_ROUTES)

        current_route = routes_to_explore[0];                // Examine first route
        route_length = getLength( current_route )
        route_colors = getColors( current_route )

        ordered_routes = {}  // Map to store lengths of paths traced by each color

        FOR current_color IN route_colors:

                IF inExcludeList( current_color ):
                        SKIP

                stretch_iterator = route_length + 1   // Perform lookahead for current group

                WHILE stretch IS LESS THAN END:

                        color_groups_at_marker = getColorGroupsAt( stretch_iterator  )

                        //stop lookahead if new color encountered that is not a zero group
                        IF NOT current_color EXISTS IN color_groups_at_marker:
                                IF NOT color_groups_at_marker EQUAL zero_color_group:
                                        BREAK

                        stretch_iterator  ++

                // After lookahead for current route and current color is finished,
                //  store color with key as the length of current lookahead stretch
                lookahead_stretch = stretch_iterator  - route_length
                ordered_routes[ lookahead_stretch ] = current_color

        // Examine color keys in descending stretch order
        FOR color_stretch IN descendingKeyOrder( ordered_routes ):

                IF color_stretch LESS THAN OR EQUAL TO 1:
                        //Not significant change, dead end route.
                        SKIP

                color_for_stretch = ordered_routes[color_stretch];

                // Create a new route, based on the current route with the lookahead color
                // added the same amount of times as the color stretched.
                new_route = clone( current_route ) + addcolor( color_for_stretch, color_stretch)

                IF length(new_route) EQUALS END                      // Found a complete route
                        complete_routes.add( new_route )

                ELSE:                          // Otherwise, add new route to active search
                        routes_to_explore.add( new_route )

        // Remove current route, now expanded by new routes
        routes_to_explore.remove( current_route )
```

Text 4: A Best-First Haplotype Pseudocode Representation*

Only one route can be chosen as the best from the complete routes, and this is determined by sorting the complete routes by ascending number of sets and picking the first route, i.e. the route that has the fewest unique colour groups and number of meioses.

1.7.5 Optimization

Several optimizations are undertaken to reduce the overall complexity and excess system resource usage of the algorithm; and these can be broken down into modes attributed to stages of processing that occur prior to, during, and after the search algorithm has run its course.

Prior Processing: Step Contraction

The first speedup is introduced during the first pass of procedure, where the overall number of nodes that the search algorithm has to evaluate during the lookahead section is reduced.

Previously, in order for the algorithm to perform a lookahead it had to step through each marker in order to determine whether the colour group under consideration existed at that locus. By adapting the *child2parent_link* function to store a sliding window of the colour groups it processes, a very rudimentary lookahead can be performed within the same pass used to prime the colour groups.

This effectively means that prior information upon the colour groups can be gained and utilizedby the A* search, such that nodes are tailored so that the algorithm can pass over known stretches without having to crawl through them first.

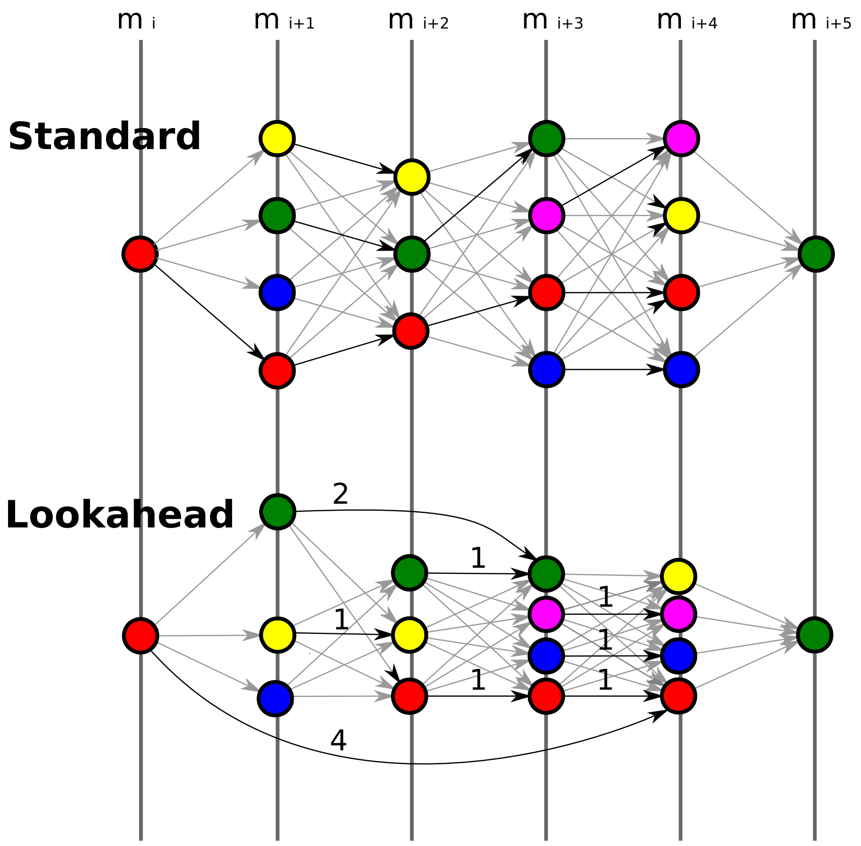

Figure 1.8: Step Contraction, assisting in the A search algorithm by reducing the number of steps needed to be evaluated, e.g. R-R-R-R-R-G under the Standard priming technique (Top) can be reduced to R-R-G under the Lookahead priming technique.*

Figure 1.8 illustrates this quite effectively; where the information contained within the set has not been transformed under the Lookahead arrangement, only that the more optimal paths are better defined. For the other less optimal paths, nothing has been changed: R-Y-G-B-R-G remains the same under both representations.

The gains in this technique may seem small, and indeed the total number of nodes and edges between the Standard and Lookahead representations in Figure 1.8 differs by only 1. However, A* is a best-first algorithm, meaning that only the leading paths in each expanding iteration of the frontier are considered. It is these leading paths that benefit the most from this step contraction, since they are the ones ultimately will have to step through the most.

For a sliding-window of size *s* stepping through a haploblock of estimated size h/m for *m* meoises and *h* initial founder length, this produces a linear reduction on the order of h/sm .

Runtime Processing

Debug and Release

The A* algorithm is actually first defined in the function *a_star_bestfirst__DEBUG*, which logs to console every step of the analysis for later inspection. This is useful for debugging purposes, but is impractical during standard processing because the JavaScript runtime environment must queue these output messages to be printed in a timely manner, expending system resources.

The flexibility in the JavaScript Object model means that attributes can be bound to a given object during runtime and do not need to be explicitly defined before[27]. Since functions and objects in JavaScript are implemented the same way, this means that functions can be defined at runtime too.

A new optimized "release" function *a_star_bestfirst* is defined that takes the content of the debug function and truncates all logging references in order to boost performance. This also means that two varieties of the function only exist when required (i.e. during processing), and that the optimized function does not need to be manually re-written every time the original function undergoes a revision.

Homology Path Bunching

This extra feature is less a performance optimization and more an analysis optimization. As an analysis progresses through a chromosome, the top *N* paths begin to "bunch" together in such a way that the *MAX_ROUTES* limit could compromise the admissibility of the search, if the similarity between active paths becomes too homologous as $N \rightarrow MAX\_ROUTES$.

27Several other languages incorporate similar functionality (C++ via virtual functions) but with more rigid parameters.

To prevent this bunching behaviour and encourage a more diverse selection of paths, the number of active paths *N* only contribute towards the maximum route limit if they do not share the same length score at given iteration. This essentially changes *MAX_ROUTES* to *MAX_TIERED_ROUTES* such that unique sets of active routes are considered.
For example, a *MAX_ROUTES* limit of 4 would be able to give active route lengths of {32, 21, 21, 21, 15 ,12,12} instead of {32, 21, 21, 21}. Paths that have identical scores are usually indicative of a long stretch where (in the case of 21, three) colour groups maintain their colours through an ambiguous region that caters for them all.

By allowing a greater variety of path sizes, earlier branching points in the analysis are not pruned off too eagerly and provide a more diverse set overall.

Post Processing: Pointer Cleanup

After the processing is complete and the result of the best-first search is returned unambiguously for both parental alleles, a cleanup operation is performed upon the Allele class containing the data. The pointer array contained within the class was a non-compact representation of all the possible colour groups that could propagate to a given individual through accepted models of inheritance, and was operated upon via the A* search.

Once the search is fully complete for all members of the pedigree, the pointer arrays have served their purpose and are no longer required. The function removePointers in the file pointer_cleanup.js sweeps down through pedigree and initialises the haplogroup_array with a compact signed int8Array that stores the best path for each individual-chromosome. The pointer array is then requested for a delete by the JavaScript garbage collector.

The memory saved from this cleanup should not be understated. For a typical 1000 marker chromosome stretch with a maximum of 8 possible founder allele groups (from 4 founders) where each allele group is a (64-bit) reference to a group, this produces a total of (1000 * 8 * 64 = 512000 bits =) 64 kilobytes per chromosome. By compacting the colour group data into an 8-bit array, the total is shrunk down to (8 * 8 * 1000 = 64000 bits =) 8 kilobytes per chromosome.

It is arguable that the only savings being made are at the kilobyte level and even the most modern platforms by default sport at least 1GB of memory, however the data is now structured in contiguous parts of memory instead of dotted around via pointer arrays and the JavaScript runtime environment can now free up memory and run under a smaller more optimized region of the RAM.

1.7.6 X-Linked Inheritance

As shown in the Autosomal Inheritance implementation on page 47, parental alleles are represented as {*p1 p2*},{*m1 m2*} pointers which each represent a possible array or founder allele "colour" groups. The valid child allele {*c1 c2*} configurations are shown in Table 3 below.

Model	**Parental Alleles**		**Allowed Child Combinations**	**Total**
	Maternal {p1 p2}	*Paternal {m1 m2}*	*{c1 c2}*	
Autosomal	{a1 a2}	{a3 a4}	[a1 a3], [a1 a4], [a2 a3], [a2 a4]	8
X-linked (Female)	{x1 x2}	{x3 y1}	[x1 x3], [x2 x3]	4
X-linked (Male)	{x1 x2}	{x3 y1}	[x1 y1], [x2 y1]	4

Table 3: Three separate methods of inheritance. Parental alleles given in curly brackets denote ordered alleles. Child alleles are mutually exclusive of different parental types, and are given in square brackets to denote possible combinations (e.g. [a b] = {a b},{b a}).

It can be seen that the pedigree being dominant/recessive has no effect in the way that the alleles are inherited, the only deciding factor being X-linkage paired with gender-specific cases. Even if the pedigree is consanguineous, the model still holds because any founder blocks that find their way into both alleles of an individual, did so because their colour groups were contained in the separate maternal and paternal alleles through different paths of descent.

Under the pointer model, X-linked inheritance is not a very involved process since it acts merely as a more constrained version of the autosomal model under different gender designations. The only unknown exists in the X-linked male model, where knowledge of which allele indicates the Y chromosome is required prior to evaluation.

Initially it was a hard-coded requirement in the input file that the second male allele was the Y-chromosome, but this restraint was later removed and resolved in the software via the function *detectYalleles* in *filehandler.js*.

At this current time of writing, Y-chromosome capable genotyping exists but is not utilized in most linkage software[28], which ignores it altogether and designates the entire chromosome to an array of zeroes. The function simply detects these zero alleles in the males when parsing the pedigree, and stores the Y-chromosome as the second allele in the Pedigree class.

In the event that Y-chromosome capable genotyping becomes more widely used, the *detectYalleles* function will need to perform further probing. This will not preclude the rest of the analysis, since the Y-allele will still be assigned to the second allele in males.

28 Allegro, Genehunter, and Simwalk have not been updated since 2003. Genotyping chipsets have greatly increased in density and coverage since then.

Comparison with HaploPainter (<XXX: Probably should go in results>)

One of the core reasons for HaploHTML5's conception was the lack of X-linked inheritance within the more commonly used tool HaploPainter, where haploblocks would be rendered improperly. This was at first thought to be due to the linkage software that performed the haplotype reconstruction assigning the wrong allele indexes to untyped individuals who were reconstituted in the program, but closer inspection revealed that allele indexes were in fact correctly assumed and that the problem lay in the haploblock resolution software.

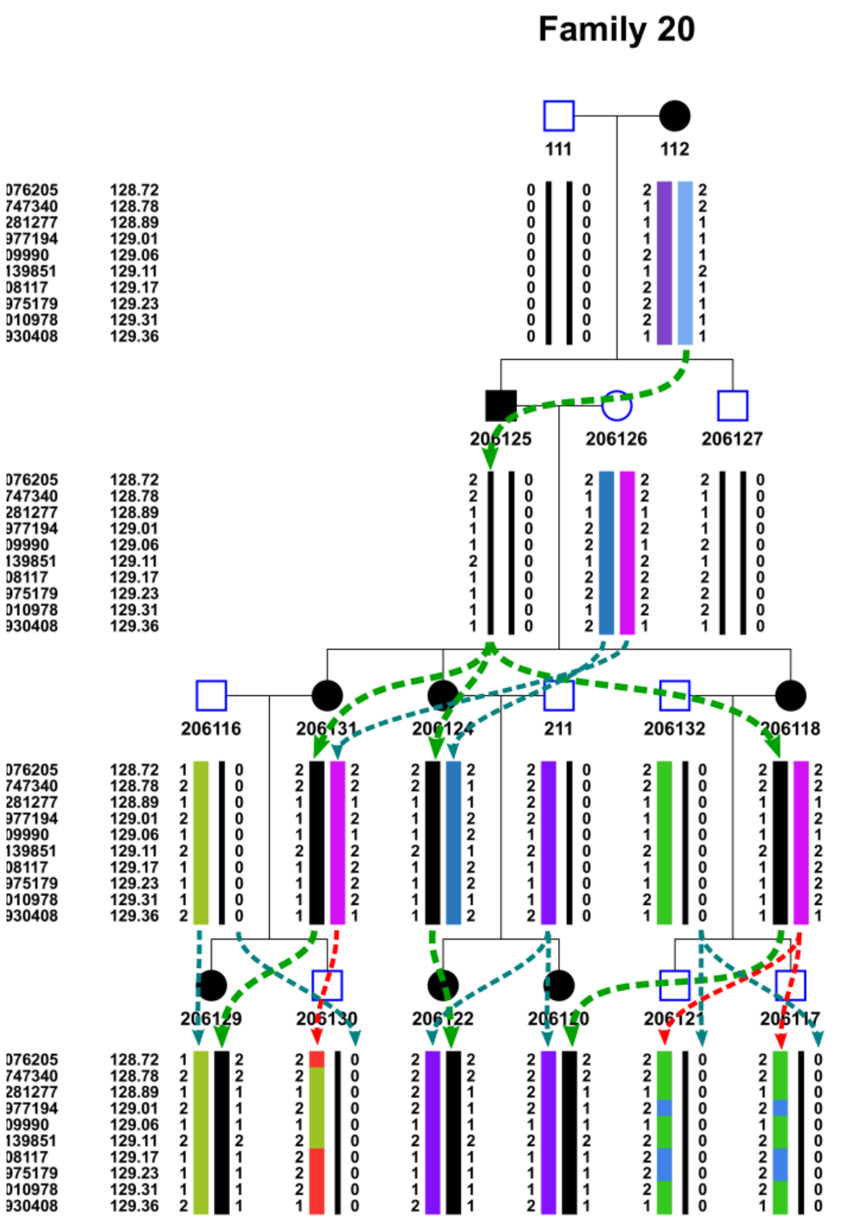

Figure 1.9: Minimum working example of an X-linked Dominant pedigree (source attached in Appendix). Inconsistent colouring within alleles depict HaploPainter's haploblock resolution failing in the case of X-linked pedigrees. Overlayed arrows depict true inheritance paths; black disease allele (green-dashed), correctly inherited alleles (blue-dashed), incorrectly inherited alleles (red-dashed).

Figure 1.9 shows a typical X-linked dominant pedigree as rendered by HaploPainter. There are several issues highlighted in this example:

1. **Non-zero alleles assigned zero-allele blocks** – This is shown in the left allele of individuals 206125 and 206127, which are represented as narrow blocks normally reserved for the zero-alleles.

2. **New haploblock group assigned to non-founder –** 206130 shows the appearance of a red block that is neither inherited from parents, grand-parents, or great-grandparents at the current locus. This is a misalignment because an inspection of the allele numbering shows that the block should be inherited from the maternal pink allele (as shown by the red-dashed arrow).
3. **Impossible recombinations –** 206121 and 206117 exhibit multiple recombinations over extremely small genetic distances. For reference, a distance of 1cM would expect 0.01 recombinations. The genetic distances here are on the order of 0.01 cM, which makes multiple recombinations very unlikely. The red-dashed arrows once again show the true path of inheritance, where both individuals inherit their mother's pink allele as confirmed by the allele numbering.
4. **Non parental-offspring inheritance –** The appearance of the blue haploblock in 206121 and 206117 is troubling, since only the mother (206118) could inherit the allele from the founder (206126) and yet doesn't. Even more concerning is the apparent mixing of non-sister alleles in the offspring. Recombinations take place within the parent first before being transmitted, which creates clear maternal and paternal alleles in the offspring. 206121 and 206117 appear to show the green (paternal) allele recombining with the blue (maternally inherited) allele which should be impossible under any inheritance model.

It is likely that the method that HaploPainter uses to assign allele groups does not distinguish between bound and uncertain genotypes, very liberally pushing all possible founder allele groups onto both child alleles without any mutual-exclusivity, effectively resulting in a "mixed bag" of possible colours. Though this gives the algorithm more leeway to find any solution instead of terminating with errors, it does (in this instance) produce incorrect results.

1.7.7 Pedigree Rendering

A pedigree is populated by reading in a file in the standard MAKEPED linkage format (see Background section XXX) and storing the data in a containing structure (Person class in this case). However this specifies information about each individual in the pedigree and not their relationships to one another.

The function *connectAllIndividuals* (pedigree.js) iterates over all stored members and creates connections between each: parent is added to child, and child to parent, with parents being added as mates to each other. Each family is stored in its own separate map so that different families are segregated into their own pedigrees.

The process of actually drawing the pedigree is a complex process because as mentioned in Section 1.7.1 traversing through a graph involves expanding nodes and all nodes that are connected to them.

Consider now a graph where there is no longer a single root node, but instead several, as in the case of a pedigree with many founders. If node expansion was performed upon each of these individuals, we could descend through the graph by perform a breadth-first search upon all offspring.

A problem will quickly arise however where individuals who exist in the descent path of multiple founders will be evaluated more than once and lead to duplicity in the data if not carefully handled.

The peeling method that Elston and Stewart devised (as mentioned in Background, page <XXX>) was considered as one solution; determining pivot individuals and treating them as special edge cases to be separately handled when drawing the pedigree. However, previous experiences in handling edge-cases (see page 38) had not been straightforward and so a more general solution was derived instead.

The resulting function *populateGrids_and_UniqueObjs* (init_graph.js) required iterating through all individuals and populating an array that stored each generation, as well as populating a map which stored unique nodes and vertices associated with pedigree members and their relationships as discussed below.

Populating and Structuring the Pedigree

The grid generation is contained within the class *addFamMap* (init_graph.js), which asserts that each pedigree has a single randomly chosen individual to act as the root node. The sub-function *addNodeArray(person, level)* descends through all relationships associated with that individual (mates, children, mother, father) recursively until there are no more relationships to expand upon.

Unique Graphics Map

Upon each recursion, a shape representing that individual's gender and affectation is inserted into the graphics map with the individual's identifier being the accessing key. If trio relationships are also detected, then the trio is added as the combination of two unique lines:

- **Mateline –** A straight line connecting two individuals who share offspring. Their unique key is paternal ID and the maternal ID in that order.

  e.g. 11 (father) and 12 (mother) → "m:11-12")
- **Childline –** A right-angled line that connects an offspring to the centre of the Mateline of that offspring's parents. Siblings share the same centre (or 'anchor') but different childlines. Their unique key is the Mateline ID and the offspring's ID in that order.

  e.g 14 (offspring of 11 and 12) → "c:m:11-12-14"

Under this format, even if individuals are evaluated twice the map will not contain duplicate data because the order of parental gender and offspring is preserved so that the same relationship is only recorded once.

Generational Grid Array

Within the same recursion step that the graphics map is populated, the *level* parameter records the ascension/descent of the relationship being expanded. With the parameter initially set to 0 for the root individual, parental relationships are expanded with *level – 1* parameter, and offspring with a *level +1* parameter. This ensures is that not only are all siblings at the same "level", but first-cousins too.

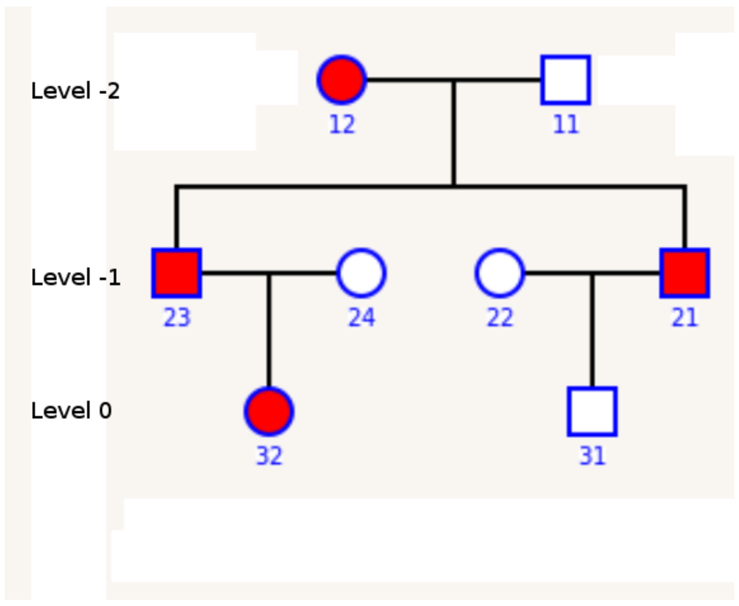

Figure 1.10: Example of pedigree levels and recursion

Consider Figure 1.10 which shows three levels for a group of individuals as determined by recursive function. If we pick 32 as the root individual with a level of 0, then we can trace a route first-cousin 31, via 32 → 23 → 11 → 21 → 31:

- Start at Root: 32, level = 0
- Expand to Parent (level – 1): 23, level = -1
- Expand to Parent (level – 1): 11, level = -2
- Expand to Offspring (level + 1): 21, level = -1
- Expand to Offspring (level + 1): 31, level = 0

Thus we have determined that 32 and 31 share the same level and can be drawn at the Y-position in the pedigree as shown in the figure. By expanding upon all individuals, we can very easily deduce the individuals who should constitute a given generation and represent our pedigree accordingly.

The Separation of Graphics and Data

The information generated and stored at this stage follows a strict Model-View structure, with the graphical data to be drawn on screen being held entirely within the global graphics map *unique_graph_objs* which is accessed solely through the static library class *uniqueGraphOps* via getters and setters for adding/removing nodes and edges.

This is divorced completely from the the pedigree data that pertains to individuals' information storage (name, gender, parents, offspring, mates) which is held entirely within the global data mappings *family_map* which is likewise accessed solely through its own static library class *familyMapOps* via its own getters and setters for adding/removing individuals.

It cannot be stressed enough that the Model (*familyMapOps*) and the View (*uniqueGraphOps*) do not overlap, i.e. one does not have pointers that reference entries in the other or vice versa. This ensures clean code management and simplifies tasks drawing tasks since the graphics should ultimately be a disposable representation of the pedigree data. The deletion of an individual's graphical node should not also prompt the deletion of that individual's pedigree data.

Determining Penetrance Model

The right penetrance model is key to running the correct analysis upon the pedigree, and the function *determinePedigreeType* (pedigree.js) was written to determine whether the pedigree was X-linked/Autosomal, and Dominant/Recessive.

Though Table 3 showed that the analysis does not depend on dominance/recessiveness, it is still useful in the context that it can be used for other programs that may require it without having to prompt the user.

The function iterates through every member of the pedigree and operates as follows:

- **Detecting X-linked**
  1. Calls upon the *detectYalleles* function repeatedly to count the number of males with either a single allele, or with an allele consisting entirely of zeroes. This lends some flexibility in the input format, where a single allele need only be specified for a male in a pedigree and the program will automatically detect that the pedigree is X-linked, and will generate a second Y-allele for that male (consisting of zeroes).
  2. The function asserts that the pedigree is X-linked if the **all** males in the pedigree are equal to either: the number of males with a single allele, or the number of males with an all zero allele, or any combination of the two.
- **Detecting Dominant**

  Operates on a much simpler notion of descending through each generation and counting the number of affecteds. If the number of affecteds in each generation is non-zero, and the number of affected generations is the same as the number of generations; then the pedigree is dominant.

Note: Autosomal and Recessiveness are determined by inverting the results of these two methods.

Detecting Consanguinity

Couples who share either parents, grand-parents, or other immediate ancestors are said to be consanguineous and their mateline indicates this via a double line. Detection of this is actually also implemented by a (much smaller) A* best-first search to find the highest common founder between two mates.

The function *hasConsanguinuity(person1, person2)* (init_graph.js) is outlined concisely in the following pseudo-code:

```
FUNCTION hasConsanguinuity( p1, p2 ):

    complete_routes = []
    routes_to_check = [ p1, p2]

    WHILE routes_to_check NOT EMPTY:
        // get first person from active routes array and remove them
        person = removeFirst( routes_to_check )

        // No parents, terminate active route
        IF NOT ( hasMother( person ) AND hasFather( person ) ):
            complete_routes.add( person )
            SKIP

        // Otherwise add parents to search
        routes_to_check.add( getMother( person ) )
        routes_to_check.add( getFather( person ) )

    // Check for duplicates in complete routes
    RETURN duplicatesExist( complete_routes )
```

Essentially the search begins with the two individuals *p1* and *p2* who are mates. The search runs under the assumption that their ancestors are not related, and so recursively expands up their parents and their parents-parents, etc. until there are no more parents to evaluate.

This populates a global list of founders as derived from separate “routes” starting at *p1* and *p2*. If *p1* and *p2* do not share any of the same founders then they are unrelated. However if they do share founders then there will be duplicates in the global list of founders, because shared founders will be discovered more than once from the two separate starting points.

Line Management and Node Position Handling

The graphics map contains pointers to all the shapes and lines to be drawn on screen. As the user manipulates these node's positions the relationship lines associated with them must also have their positions updated.

As previously stated, Matelines are bound to the nodes of two individuals whose relationship they represent, and Childlines are bound to the anchor points of Matelines and the node of the depicted offspring .
KineticJS creates shape nodes easily enough (squares for males, circles for females, and diamonds for unknowns), and shapes are coloured to represent affection (red for affected, white for unaffected, and grey for unknown).

KineticJS also has a Line class that can represent a straight line between two points, or a path through multiple points under a variety of different styles. However, it does not have a Connector class that can automatically update a line connected between two objects, such that the movement of one object automatically updates the position of the line.

As a result, line management must be performed manually via graphics map. Essentially upon the detection of a single node being moved[29]:

1. All matelines associated with that node must have their positions updated.
2. All childlines associated with each mateline moved in the previous step must also have their positions updated.
3. If the individual that the node represents is not a founder, then their own childline must be updated.
4. Draw updated positions onto screen.

29 Thankfully nodes can only be moved one at a time thanks to there being only a single mouse pointer. Multi-touch mobile devices may present problems in the future however.

Given that these steps are performed repeatedly in real time until the node is no longer being moved, it may be seen as a costly process. However, it is not recursive (i.e. the positions of grand-parents and grand-children do not need to be taken into consideration), so at most only two generations of childlines are being updated.

Grid Snapping and Family Grouping

In order to reduce some of the draw operations when moving nodes in real-time, a snapping grid was implemented so that nodes cannot be placed just anywhere but must fall in line with some hidden horizontal and vertical guideline intersections; essentially "a grid".

Dragging a node between the points in this grid will not prompt a redraw, where redraws will only occur when the node is locked into one of the points on the grid.

The grid itself is not global to the screen, but is family group specific, where the graphics of each pedigree are contained within their own separate group. These family groups (and their containing nodes and edges) can be moved freely without any snapping taking place. It is only the movement nodes within these groups that are subject to grid snapping, and the grid is relative to placement of the family group.

Sibling Level Balancing, and Mate Swapping

Another advantage of the grid snapping feature is that it facilitates the user placement of first-cousins and siblings at the same level as shown in Figure 1.10 (Page 63).
To extend this functionality, sibling level balancing was introduced to overcome the tedious task of having to individually move all siblings to the same level.

Siblings positioned at different levels does not makes sense in a generational context and looks unsightly, and it became clear that it would be beneficial to move all siblings automatically. The dragging of a single node prompts the vertical movement of all nodes sharing the same mateline anchors.

In a similar fashion, it also looks unnatural to have a mother and father at two different levels and so parents are also locked vertically such that the movement of one also updates the other. Parents also have the added feature of being dragged as a single unit if they are brought together. This may seem as if it would prevent mates from moving past each other in a certain direction, but automatic mate-swapping occurs to prevent this, allowing the free ordering.

Pedigree Creation (PedCreate View)

Pedigrees can be created from scratch without having to load a pedigree file. When creation mode is enabled individuals can be added to families, joined together as mates (with available choices highlighted in red circles), and joined as offspring (with available anchor points appearing within matelines) as depicted in the figure below.

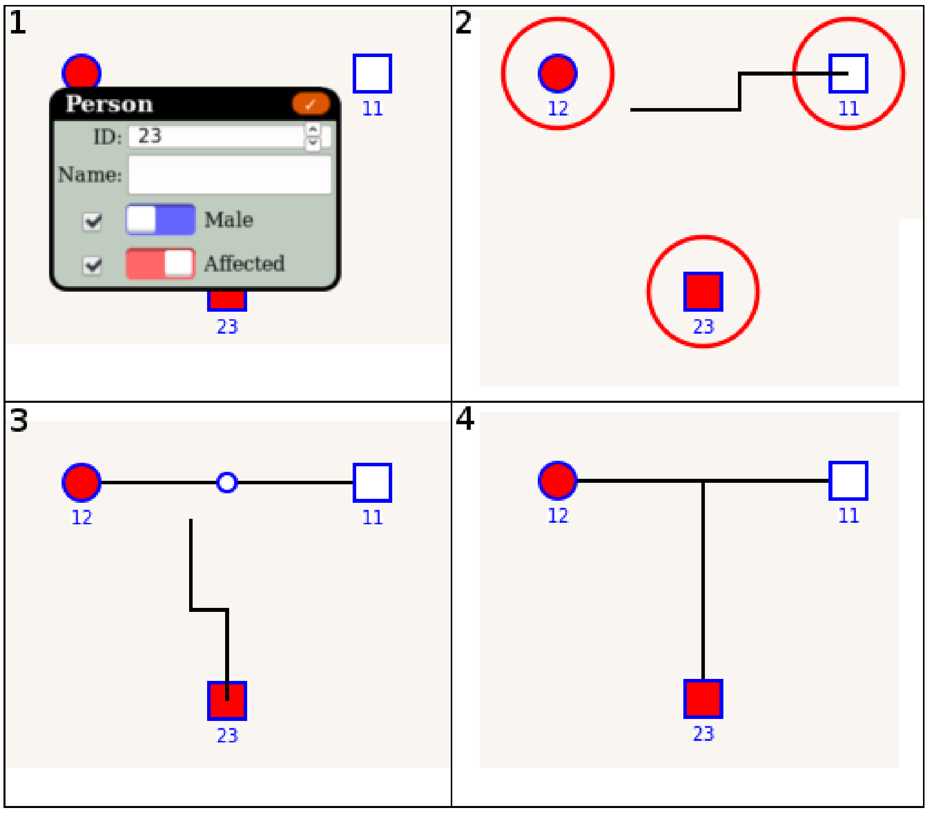

Figure 1.11: The four stages of creating a pedigree: (1) Adding individuals, (2) Joining mates, (3) Joining Offspring, (4) Completing a trio.

Once complete, the pedigree can exported as standard MAKEPED file, as well as optional non-standard tags which contain metadata such as individual names (i.e. not just IDs), and position upon rendering which are interpreted by the application.

Selection Mode (Selection View)

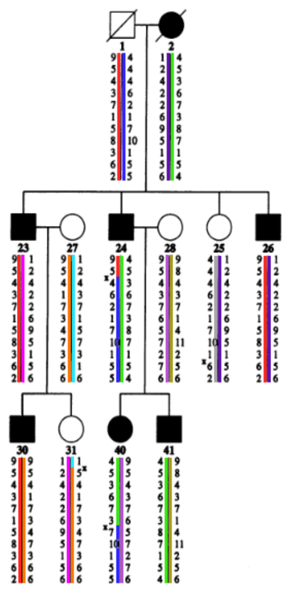

Figure 1.12: Example haplotype analysis spanning three generations and 11 markers

Prior to HaploHTML5, analysing the haplotypes of individuals required repeatedly scrolling up and down the screen because individuals at different generations would be represented vertically.

This is not a problem for small loci, and is even beneficial when trying to see the path of descent. However, examining larger loci becomes difficult when trying to compare allele indexes because the user needs to mentally recount the position of a given marker at each generation.

HaploHTML5 does away with this notion by allowing the user to freely select whichever individuals they want to analyse, whether from the same family or not, and then transitions the individuals onto a side-by-side level view so that haplotypes at a given locus can be be compared across all generations simultaneously as shown in Figure 1.13.

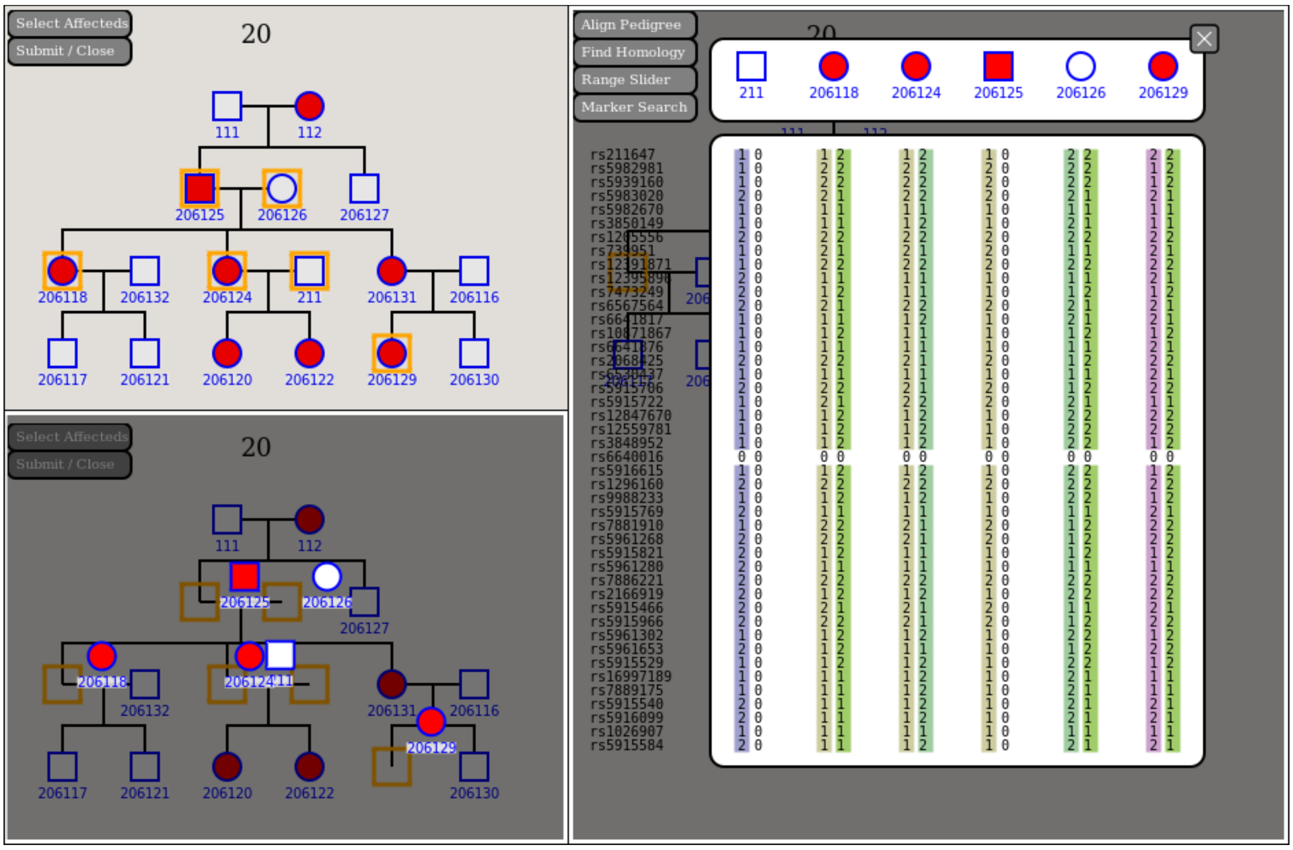

Figure 1.13: Selection mode in three stages; (Top Left) User selection, (Bottom Left) Transitioning to side-by-side view, (Right) Comparison mode

Haplotype Navigation (Haploblock View)

Haplotypes are initially displayed in a viewport restricted to a particular locus of interest for a default stretch of 30 markers. The user can then move through haplotypes using this viewport by one of:

1. **Scrolling the Mousewheel** – Haplotypes are moved up/down by a single marker for each mouse wheel increment.
2. **Dragging and Dropping** – Haplotypes can be grabbed directly by the mouse and the view can be shifted up/down by multiple marker stretches.
3. **Range Slider** – A slider can also be used to quickly navigate up and down the chromosome. This is by far the quickest method, and has the added bonus that the viewport size can be adjusted by changing the handles of the slider, as well as a quick overview for the scale of

the chromosome being represented by the sliders boundaries (see Figure 1.14). Points of recombination are best viewed by this method since quick scrolling makes coloured block changes immediate.

4. **Marker Search** – This makes use of HTML5's *datalist* element which populates a list of available markers and assists the user by allowing them to either choose from the entire list of markers via a dropdown menu, or an autocompleted sub-selection as they begin to type out a marker identifier. This method is the most effective when the locus of region is precisely known (e.g. due to linkage) and the user wants to quickly navigate to a region of interest without having to scroll for it.

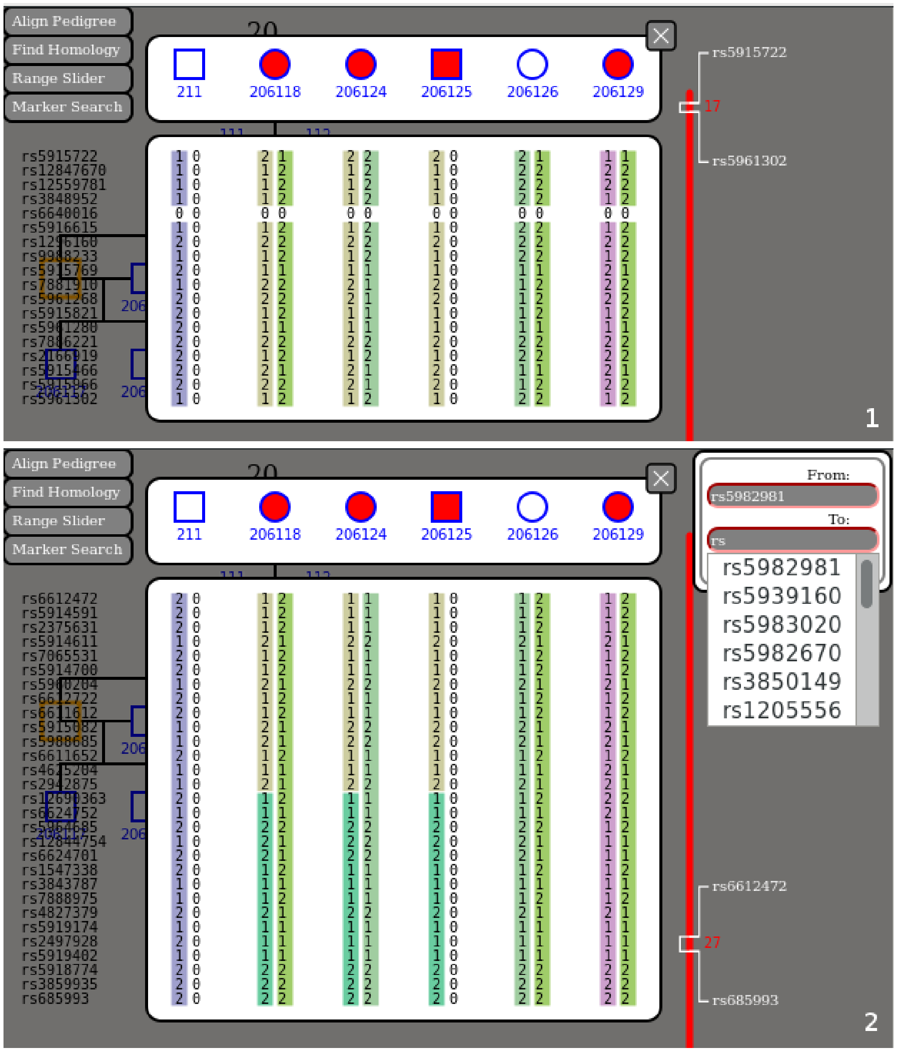

Figure 1.14: (1) Range slider and (2) Marker search with region expansion via slider handles.

Analysis Resumeability

The application makes use of the *localStorage* JavaScript implementation to cache the last used pedigree or haplotypes file, so that the user does not need to manually reselect it every time they want to rerun an analysis. The browser simply pulls the file from local storage, and meta flags in the file return to whichever view that the user was last on (PedCreate, Selection, or Haploblock).

1.7.8 Homology tools

One of the main reasons that haploblocks are analysed post-linkage is to find regions of homology pertaining to the linkage region(s). Where the Haploblock View allows the user to make side-by-side comparisons of cases and controls over multiple families, the Homology View further enables a sub-selection of those individuals to be checked for homology, as shown in Figure 1.15 below.

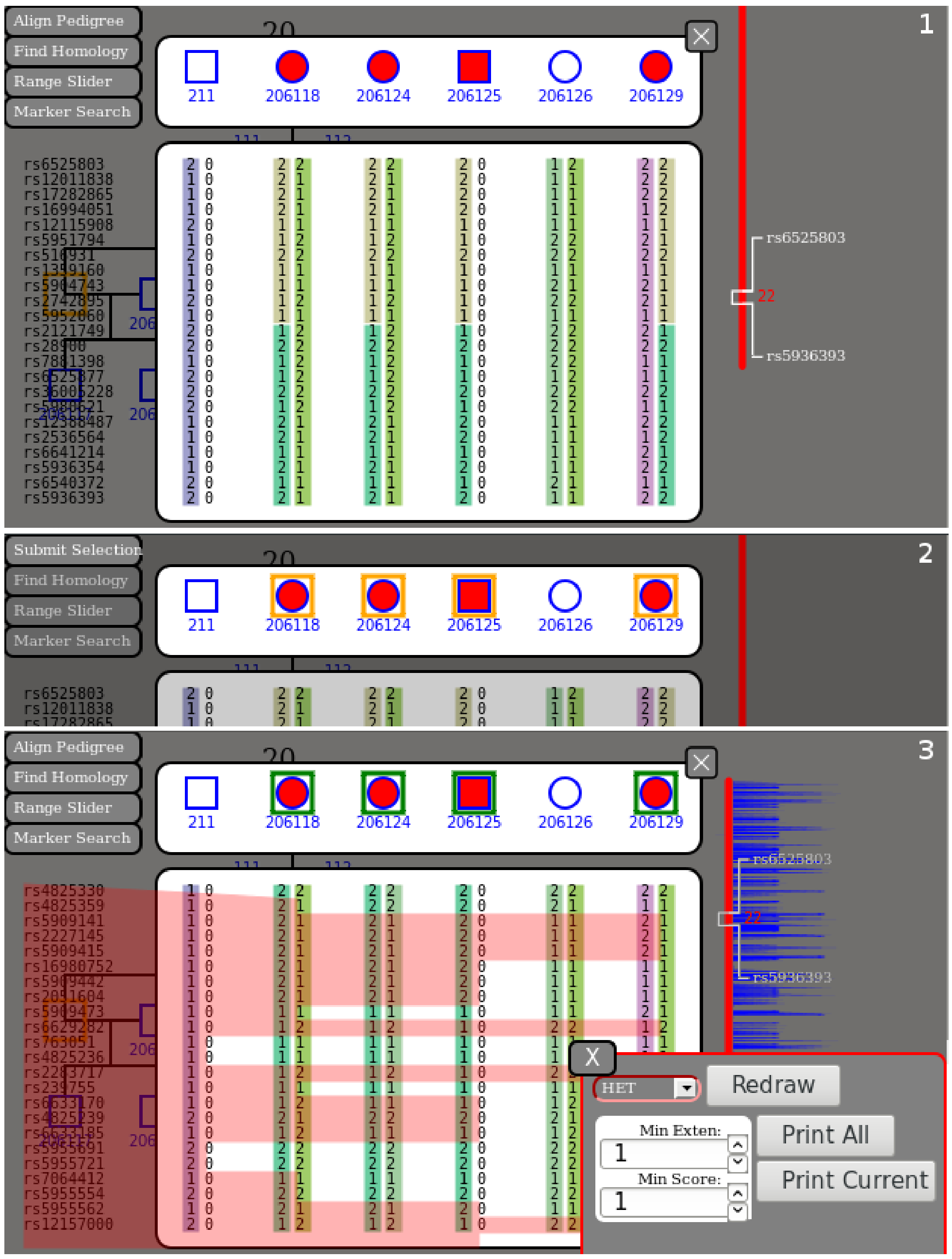

Figure 1.15: Homology Mode activation; (1) Haploview mode, (2) Find homology button pressed where user can select which individuals should be compared for homology (in this case the affecteds), (3) Homology plotted across marker range (blue) and overlayed over haplotypes (red)

Upon sub-selection submission homology, individual alleles are sorted into groups of affecteds and unaffecteds; the idea being that affected alleles that match to some model contribute to a score and unaffected alleles that match to the same model subtract from the score. Thus for a scenario where all alleles (affecteds and unaffecteds) have the same homology, a total score of zero will be produced.

Scores are calculated for three scenarios:

1. **Homozygous –** Both alleles must be identical and must match all other affected alleles to contribute to the score. If only 1 allele of an individual matches, the score will not increase.
2. **Heterozygous –** One allele must match at least one allele in all other affected individual alleles to contribute to the score.
3. **Compound Heterozygous –** Both alleles must differ and must match all other affected alleles to contribute to the score.

All three scenarios are determined in the same pass, but cannot be represented simultaneously as an overlay, so it is up to the user to select which mode they wish to view, and a redraw is required upon mode change. The user can also filter the data so that scores below a minimum score threshold are not shown, as well as a minimum extension limit so that only regions of homology spanning at minimum a certain amount are shown. This is especially useful when trying to constrain a wide linkage peak to a more narrow region of interest. Recommended threshold values are Min Extension = 2, and Min Score > 3

1.7.9 Code Minification and Obfuscation

Before the licensing of the application had been decided, attempts were made to scramble the source code such that it could be incorporated into more restricted models without privacy being a concern.

Browsers do not run closed-source binaries, and deploying a JavaScript application as such through a toolchain that would compile it into a binary would not make sense over the internet, nor would it run in-browser.

The only other option was to somehow encrypt the code itself in a manner that it could decrypted at runtime and run as a normal web application. This process was broken down into six stages:

1. **Minify the code** – Code minification is the act of taking a piece of code such as "var myVariable = 123; var myOther = myVariable" and condensing it into a smaller space such as "var a=123, b=a";. This was performed using Google's Closure compiler using the *ADVANCED_OPTIMIZATIONS* parameter which not only condensed the code by renaming symbols, but also performing compiler optimizations (see Appendix section) to reduce the code structure into a more CPU friendly format. All minified code is then consolidated into a single file for easier handling.
2. **Picture Encryption** – The Python script *to_image.py* takes in a string of text and converts it into a square image where image transformations are performed upon it. This is what the single consolidated minified code file is fed into. Two key steps occur:
   a) The code is stored as a long contiguous character array, which is then broken into 32-bit elements, each of which are then broken into four 8-bit elements that represent the four (8-bit) Red/Green/Blue/Alpha[30] channels of an image (that supports transparency).
   b) The image is encoded with a fairly simple cypher:
      1. The diagonal pixels undergo a XOR[31] operation with a value of 247, and are then shifted diagonally 22 places.
      2. All other pixels undergo another XOR operation with a value of 93
      3. Rows and columns are shuffled according to a shift map.

30 The alpha channel is a requirement to prevent WebKit powered browsers (Chromium, Safari) from changing RGB values whenever the alpha is unset.
31 A XOR operation is a reversible bitwise function that essentially transforms 0 → 1, and 1→ 0.

c) The image is then saved as a lossless PNG file in order retain the pixel values determined in the previous steps.

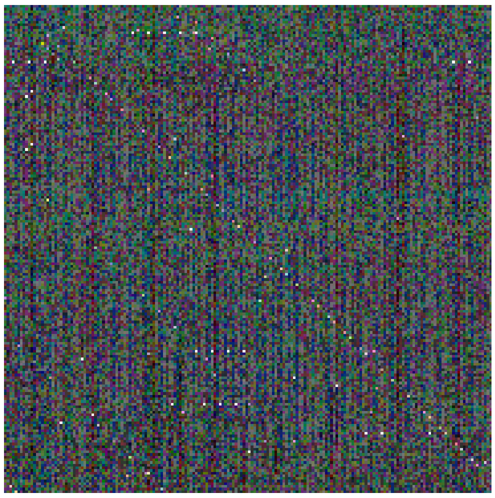

Figure 1.16: Resultant image from code minification, pixelisation, and encryption

3. **Tag swapping** – All CSS is inserted into a new index file, and Javascript to handle the decryption process is also inserted too (image_transformations.js)
4. **Picture Inclusion** – The picture is then placed into the index file in a hidden tag.

The result of all this is that when the index file is loaded, the source code of the application is contained entirely within the picture (as shown in Figure 1.16). JavaScript then decodes the image upon page load by reversing the process in step 2b. The application code is then contained within a JavaScript string, which is then used to as the body of a function created at runtime. This executes the code and the application is run without the user ever being able to see the source code from static resource files.