

Using an Interactive Genetic Algorithm for Web Page Design

by

Miyako Jones

**An Honors Thesis
Submitted as partial fulfillment of the
Requirements for the Honors Scholar Program
Concentration in Computer Science
At the University of Michigan-Flint
December 2012**

Advisors:

**Dr. Michael Farmer, Department of Computer Science/Engineering/Physics
Dr. Maureen Thum, Honors Director, Department of English**

PREFACE

When I considered transferring to the University of Michigan-Flint, I attended one of the information sessions organized by the Office of Admissions. The honors program was mentioned and the program-funded off-campus study opportunity was highlighted. I thought it would be great to be a member of the honors program, but I doubted my ability to be admitted and also my ability to complete the off-campus study requirement. I had never lived away from home for more than two weeks at a time and had only briefly considered going away to college. Over time, I came to think differently. I convinced myself during my Sophomore year to apply to the Junior-Senior phase of the honors program as a transfer student, never believing that I would be accepted. When I was accepted, and after I completed the introductory course with an A, I realized that I did have what it took to complete a rigorous course of study.

As I searched for locations where I could complete an off-campus study in computer science, I decided to study abroad in Japan. This idea was primarily motivated by my love of Japanese animation and graphic novels. Unfortunately, the vast majority of summer programs that take place in Japan are aimed at Japanese language students. The only technology-related program I could find that I was eligible for was through Michigan State University's Japan Center and it cost far too much for me to afford even with the maximum amount I could request from the honors program. I came to the conclusion that I should wait a year with the hope that another opportunity would come along.

Over Christmas break in 2008, I suddenly decided to apply to summer programs. I am not sure what motivated me. It could have been the fact that there is little opportunity

to do something interesting in the field of computer science in Flint, MI as there are few technology companies and the computer science program at UM-Flint is small. The application process for most of the programs was very detailed and time consuming. They wanted essays, transcripts, and letters of recommendation. Some of them also required a résumé or a statement of research interests. I asked my academic advisor, Dr. Suleyman Uludag, for a letter of recommendation and also asked Dr. Stephen Turner, another computer science professor whose programming course I was taking at the time. I applied to five programs and was accepted to all. Four of them were summer research programs sponsored by the National Science Foundation (NSF) while the other was an internship in Washington D.C. I was also selected for a phone interview by a representative of Sandia National Laboratories, but I had to decline as I had already made the decision to go elsewhere. I was encouraged by several people to accept the offer from the Georgia Institute of Technology (Georgia Tech), not only because it is a prestigious university, but also because the program would pay for travel, accommodations, and give me a stipend much higher than any of the other programs. The acceptance e-mail from Georgia Tech came in early March, approximately a week after I submitted my application materials. The starting date of the program was May 26th and it lasted for ten weeks.

As I had never lived on a college campus, I had no idea what I needed to bring besides clothing, toiletries, sheets, and towels. My mother and I both did research, and from this came a list of definite and "just in case" supplies. The coordinators of the Georgia Tech Summer Undergraduate Research Experience in Engineering/Science (SURE) program sent information over the next month or so about how to prepare, where

I would be staying, and other things, but nothing about the research I would be conducting. Since it was close to the end of the semester, I told the honors director, Dr. Maureen Thum, that I did not have enough time to write the off-campus proposal and that I would be fine without financial assistance from the honors program. What I had not known is that a proposal must be written and accepted by the student's advisor, Dr. Thum, and the Honors Council in order for the trip to qualify as an off-campus study experience. I contacted one of the SURE coordinators, Dr. Gary May, with a request for more information and he gave me the e-mail address of Dr. Ayanna Howard, who was to be my faculty advisor. She is an associate professor in the School of Electrical and Computer Engineering. I told her that I was thankful to be chosen but I doubted my ability to assist her with her research. She explained that she selected me because of my experience in computer science and my portfolio website that is hosted by UM-Flint. I was also contacted by Dr. Leyla Conrad, who is the second coordinator of the SURE program, and she described the living arrangements and told me what I would need to bring. I lived on campus in an apartment with three other program participants and those of us who were not Georgia Tech students were each assigned a guest photo identification card. It gave us access to the recreation center, certain research buildings, and the library. It also was linked to an account with funds that we were to use for our meals. We later learned that Buzz Funds could also be used in the campus bookstore, in the general store inside the student center, or to pay for printing in a computer lab. One thing we were all disappointed about was the way our stipend would be distributed. We could only be paid half the money at the end of week five and the other half at the end of week ten after completing our project report and presentation.

The title of my project was "Exploring Ways to Engage Children with Cerebral Palsy." I worked in a group with two other students participating in a similar summer research program organized by Dr. Howard that is affiliated with the Advancing Robotics Technology for Societal Impact (ARTSI) Alliance. Naquasia was a biology major at Spelman College in Atlanta and Terrence was a computer science major at Elizabeth City State University in North Carolina. The first official day of the program, Dr. Howard took us to the Technology Square Research Building on the eastern edge of campus where her office and lab are located. She gave a presentation on what she terms healthcare robotics, which is the use of robotics for therapeutic or assistive purposes. We were assigned to the task of creating a robotic playmate for a child with cerebral palsy by using a robotic arm kit, Boardmaker Plus special education software, and a Little Tikes four-key toddler piano. The robot's task was to play the piano when someone used the Boardmaker software to command it. We also had a wireless access switch so that a person who had the Spastic type of cerebral palsy would be able to interact with Boardmaker. We also met our graduate student mentor, Douglas Brooks. He was a Ph.D. student in electrical engineering at Georgia Tech who participated in the SURE program a few years previous.

The three of us divided up the tasks amongst ourselves. Naquasia was in charge of building the robot since she had experience that Terrence and I did not. She was on the robotics team at Spelman and had participated in a few robotics competitions. When this task was complete, she also took on the task of researching the connection between music and children and creating songs for the robot to play. Terrence programmed the robot to respond to commands while I created the graphical user interface that would be used to

give commands. There were, of course, a few problems along the way, but we managed to overcome them all and even go beyond the initial specifications of the project.

I did most of my work in my dorm room since I spent most of my time using Boardmaker, which I was able to install on my laptop. We had weekly progress meetings with Dr. Howard and/or our graduate student mentor, Doug. There were also weekly meetings with the entire SURE group conducted by the graduate student coordinator, Ashley Johnson, and weekly seminars about cutting-edge technological research. When we were not working or attending official functions, we had the opportunity to explore the campus and Atlanta in general. As I am not the adventurous type, I spent all of my time on campus unless I went on a trip organized by the coordinators of the SURE program. I took dozens of photos of the Georgia Tech campus and Atlanta as seen from campus. I also took a few photos of our project. My photos were all uploaded to the Flickr website. I created a LiveJournal to chronicle my experience in the program that was updated whenever something interesting occurred.

I became friends with a few of the other program participants. Naquasia and Terrence, my fellow group members; Jessica, an aerospace engineering major from University of Maryland – College Park who shared the apartment with me; Tyra, a physics major from North Carolina Central University who was a resident of Atlanta; Yesenia, a chemical engineering major from the University of Puerto Rico; and Odille, an electrical engineering major from Rwanda who attends Georgia Tech. Of all the participants, I felt the most comfortable around Jessica. We connected the very first day. I also went a few places with her on campus, which is something I did not do with anyone else.

The program concluded with a series of PowerPoint presentations by the participants. We also had to write a paper on our project. Although I worked with two other people, I had to present and write my paper alone because I was a SURE student and that was one of the requirements of the program. I was worried that my contribution would be of little importance as it was part of a larger project, but I managed to make it look interesting on its own. Even the paper met the minimum length requirement (which was ten pages), but only after I included general information about cerebral palsy and information about how technology has been used in the past to help those who have it. While I constructed my presentation and wrote my paper, I also had to help write the group paper since Dr. Howard wanted to submit it to an ARTSI conference. I contributed not only information about my part, but also information about cerebral palsy in general. My presentation was immediately before Naquasia and Terrence's, so it was my responsibility to introduce our project. Doug took a video of our robot toy with his iPhone, and I had hoped it would be possible to upload it to YouTube so that others could see how our toy worked, but I did not receive permission.

Overall, the time I spent at Georgia Tech was rewarding. I was able to observe scientific research first-hand, I learned about cutting-edge technology, and I acquired a lot of knowledge about graduate school in general. Above all, I learned that it is possible for me to live independently. I believe that I would seriously think about conducting another off-campus study if the opportunity was presented to me even if I could not obtain funding from UM-Flint. I am very satisfied with my off-campus experience. The impressions that I have taken away will last a lifetime.

ABSTRACT

A genetic algorithm (GA) is a computational method used in computer science to find the best solution to a problem when conventional methods fall short. It is based on Charles Darwin's theory of biological evolution and creates candidate solutions through simulated biological processes. GAs consist of four basic processes: initialization, evaluation, selection, and recombination/propagation. They have been applied to various mathematical optimization problems such as the classic Traveling Salesman Problem and also to more practical problems such as multicore processor architecture design. An interactive genetic algorithm (IGA) is a genetic algorithm whose automatic evaluation process has been replaced by hands-on user evaluation, typically because what constitutes a "good" candidate cannot be evaluated mathematically. An IGA has been used previously in website design to present webpage design possibilities that the user may not have considered. This project was inspired by the research of Oliver et. al. from the Université de Tours in France, but incorporates more Cascading Style Sheets (CSS) scripting at the expense of HTML scripting as per current World Wide Web Consortium (W3C) recommendations.

ACKNOWLEDGMENTS

The completion of my Honors thesis could not have been accomplished without the support of numerous people, several of whom are named below.

I would like to thank the Honors Program director, Dr. Maureen Thum, for being so supportive. She offered several suggestions that helped make my writing much better than it would have been otherwise. The effort she puts into the Honors Program and her concern for her students really makes a difference.

I would also like to thank Dr. Ayanna Howard, Associate Professor of Electrical and Computer Engineering at Georgia Tech, for selecting me to work on her project. The research experience I acquired as a participant of the SURE program significantly helped with the organization and development of my thesis. Also, had I not gone to Georgia Tech and attended a seminar on the use of genetic algorithms I might not have selected it as my thesis topic since I had decided that the concept was too complex prior to my trip.

My academic advisor, Dr. Suleyman Uludag, Associate Professor of Computer Science, wholeheartedly supported my off-campus proposal and my thesis. His willingness to make time to see me and to give me advice has been greatly appreciated. I would also like to thank him and Dr. Stephen Turner for writing letters of recommendation for the various summer research programs that I applied to.

Dr. Michael Farmer, Associate Professor of Computer Science, stepped in at the last minute as my thesis advisor, something that I will always be grateful for. He provided me with countless suggestions and guidance on the organization of my thesis as well as on the topic of genetic algorithms and artificial intelligence in general. He also

wrote several recommendation letters for me during my search for a off-campus study location.

Ms. Clara Blakely of the Office of Educational Opportunity Initiatives and Ms. Mary Mandeville of the Office of Research and Sponsored Programs helped me search for alternate funding sources when I thought that I would not have time to write an off-campus proposal and get funding through the Honors Program for purchases related to my participation in the SURE program. Their willingness to help means a lot.

Last—but not least—I would like to thank my mother, Darlene Jones, for her love and support, especially during my time at Georgia Tech when I was off on my own for the very first time. She has been the loudest person in my cheering section throughout my entire college career and I would not have been able to get this far without her.

LIST OF FIGURES

CHAPTER 1

Figure 1.1	4
An example of the Model-View-Controller architecture.	
Figure 1.2	5
An example of a client-server architecture.	
Figure 1.3	6
A search graph.	
Figure 1.4	8
An example of state space search.	
Figure 1.5	10
An example of breadth-first search.	
Figure 1.6	11
An example of depth-first search.	
Figure 1.7	13
An example of backtracking.	
Figure 1.8	15
An example of hill-climbing.	
Figure 1.9	17
An example of best-first search.	
Figure 1.10	19
A graphical overview of FADSE (Framework for Automatic Design Space Exploration).	
Figure 1.11	23
A visualization of roulette-wheel selection.	
Figure 1.12	25
An example of chromosome crossover.	
Figure 1.13	26
An example of chromosome mutation.	
Figure 1.14	28
An example of the Traveling Salesman Problem.	

CHAPTER 2

Figure 2.1	32
An overview of the IGA process.	
Figure 2.2	34
Page element selectors.	
Figure 2.3	35
CSS properties with valid values/types.	
Figure 2.4	37
Bit string organization for a chromosome.	
Figure 2.5	40
The IGA configuration page.	
Figure 2.6	45
Example of property type change caused by mutation.	
Figure 2.7	47
An example of crossover point selection.	
Figure 2.8	49
More screenshots of the pre-initialization configuration page.	
Figure 2.9	51
Screenshots of the evaluation page.	
Figure 2.10	52
The finish screen.	

CHAPTER 3

Figure 3.1	54
The five IGA layout styles.	
Figure 3.2	56
Number of acceptable candidates for pool sizes 15 and 20 using thumbnails only during evaluation.	
Figure 3.3	58
Number of acceptable candidates for pool sizes 15 and 20 using both thumbnails and lightbox images during evaluation.	
Figure 3.4	60
Average fitness for candidate pool sizes 15 and 20.	

TABLE OF CONTENTS

INTRODUCTION.....	1
CHAPTER ONE:	
BACKGROUND	3
1.1. Automation of Website Development	3
1.2. Search in Artificial Intelligence	5
1.3. Search in Auto-Design	17
1.4. Introduction to Genetic Algorithms	19
1.4.1. Overview	19
1.4.2. Traditional Genetic Algorithms.....	20
1.4.3. Interactive Genetic Algorithms	27
1.4.4. Using a Genetic Algorithm for Auto-Design	27
CHAPTER TWO:	
A WEB APPLICATION TO GENERATE WEB PAGE DESIGNS	31
2.1. Approach.....	31
2.1.1. Overview	31
2.1.2. The Encoding Mechanism.....	36
2.1.3. Why PHP?	37
2.2. The Algorithm.....	39
2.2.1. Initialization.....	39
2.2.2. Evaluation.....	42
2.2.3. Recombination.....	42
2.2.4. Finishing	48
2.3. The User Interface.....	48
CHAPTER THREE:	
RESULTS	53
3.1. Experiment 1: Acceptable Candidates	55
3.2. Experiment 2: Average Fitness Level.....	59
3.3. Final Thoughts	61
CONCLUSIONS AND FUTURE CONSIDERATIONS.....	63
REFERENCES.....	66

INTRODUCTION

My interest in genetic algorithms (GAs) originated during the research for my off-campus study proposal. Dr. Ayanna Howard, my faculty advisor during the 2009 Summer Undergraduate Research in Engineering/Science program at the Georgia Institute of Technology (Georgia Tech), is an electrical and computer engineering professor whose research focuses on robotics. She has used GAs in her research at Georgia Tech and in previous research conducted at NASA's Jet Propulsion Laboratory at the California Institute of Technology.

One of the seminars we had during the program was about using GAs to forecast energy loads. If it was possible to send power to businesses and homes on demand, with the amount of power matching current energy requirements, then power stations would not have to output a full amount of power 100% of the time, which would save both money and natural resources. During the research for my proposal, I had come to the conclusion that GAs were too advanced for an undergraduate, but one of my fellow program participants had been assigned to the professor conducting the energy forecasting research. This made me realize that it was possible for me to work with GAs as well.

After making the decision to write my Honors thesis on the topic of GAs, I had to decide on a focus. Since I am also interested in website development, I decided to search for information related to using a GA for this purpose. This led to the discovery of interactive genetic algorithms (IGAs), which modify the evaluation process of a traditional GA in order to give the user direct control over the end result of the algorithm. When using a traditional GA, the user makes configurations prior to initialization; when

using an IGA, the user still configures the algorithm, but he or she also participates directly in the evaluation/selection process. When applied to website design, the user repeatedly makes selections from a group of potential designs until either a final selection is made or the user decides to start over.

There are innumerable web page design possibilities and it is impossible for the user to explore them all even when restrictions are imposed such as page layout limitations. The IGA displays only a very small sample of the possible candidates, but there is a significant amount of variety in the designs of the initial generation. Although an optimal solution is not guaranteed to be found, it is likely that the user will be presented with ideas they had not previously considered.

CHAPTER ONE: BACKGROUND

1.1. AUTOMATION OF WEBSITE DEVELOPMENT

Website development automation largely consists of back-end, or server-side, code automation. There are numerous development frameworks available for numerous programming/scripting languages, the most popular being PHP and Python. Frameworks support the agile software development methodology, which stresses flexibility and reusability, something that has become an important web development practice due to the fluid nature of the web. The predominant development architecture used by application frameworks is Model-View-Controller (MVC), which is meant to simplify implementation and increase code reuse (Pressman and Lowe 284).

The agile development methodology is based on a set of 12 principles adopted by the Agile Alliance. The main focus of the principles is the delivery of working software as early and as often as possible. Other important concepts include working closely with all stakeholders, teamwork, and continuous improvement (Pressman and Lowe 15-16).

MVC separates web applications into three layers: the content, or the model; the user interface, or the view; and the implementation, or the controller (see fig. 1.1). The model not only contains content such as text and video, but it also contains references to all external data (such as the data stored in a database) and all associated processing logic. The view's purpose is to present the content and provide a way for the user to interact with it. The controller manages access to the model and the view as well as coordinates the flow of data between them (Pressman and Lowe 285).

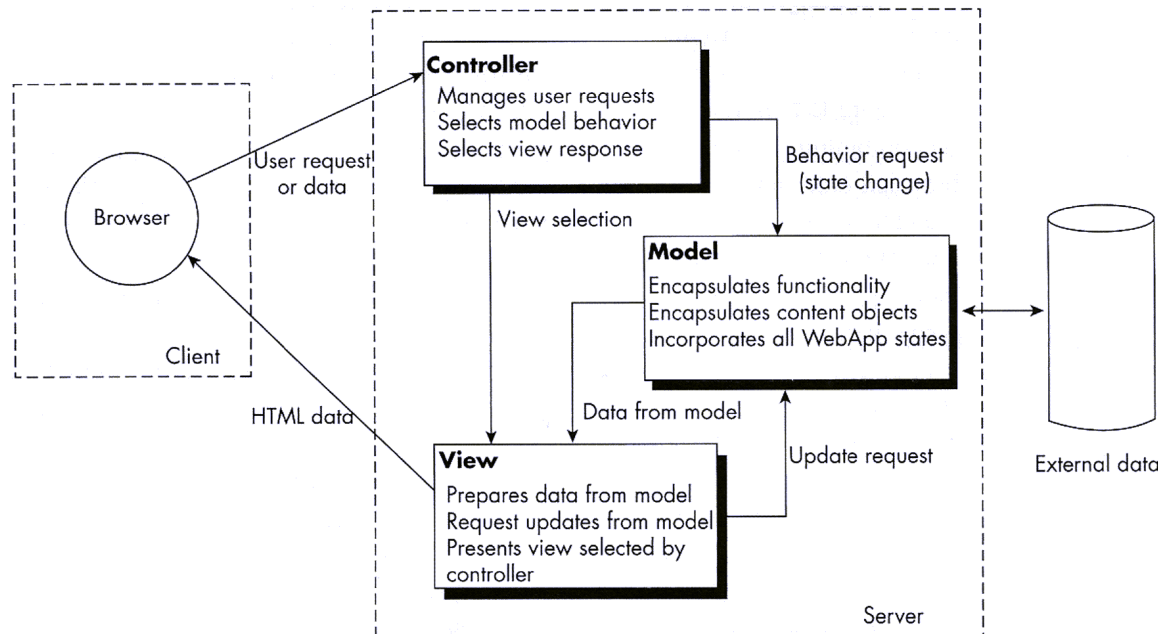


Fig. 1.1. An example of the Model-View-Controller architecture (Pressman and Lowe 285).

Website development frameworks are typically stand-alone, but they can also be bundled within a content management system that manages the content objects of the website. A framework consists of pieces of source code that a developer can use as a foundation in order to rapidly create his or her own web applications (Porebski et al.). This is unlike a library, which is external code that is called from a developer's application. What makes frameworks powerful are their use of design patterns, or "general solution[s] to ... commonly occurring problem[s] in software design" (Porebski et al.). MVC is such a pattern. Though frameworks can be useful, they are not appropriate for all types of projects. Applications that make use of a database (such as social networking sites and online stores) are ideal candidates; however, a highly specialized application or an application whose content is primarily static will not

benefit due to such things as limited customization ability and the amount of overhead a framework will add (Porebski et al.). Figure 1.2 is an example of a client-server architecture that makes use of a database. The client is the user's computer while the server is the computer that the client requests web pages from.

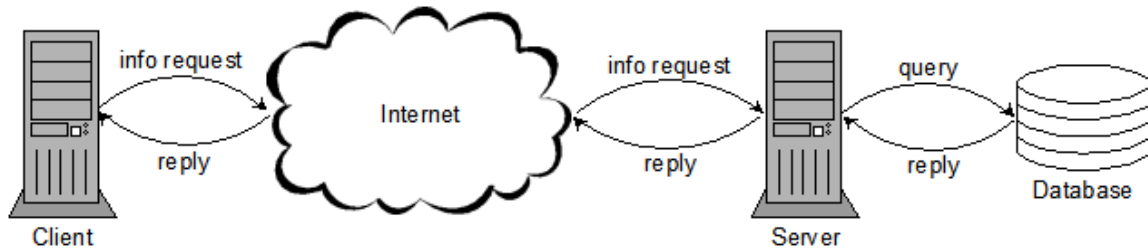


Fig. 1.2. An example of a client-server architecture.

There are frameworks available for front-end, or user interface, development as well, the vast majority of which are designed for Cascading Style Sheets (CSS). Like back-end frameworks, they are intended to enable the rapid development of websites by using built-in code as a foundation. Unlike back-end frameworks, the majority of them are not based on a development pattern.

1.2. SEARCH IN ARTIFICIAL INTELLIGENCE

The process of searching for a solution is a universal problem solving method. In artificial intelligence, *state space search* is a conceptual tool used to design intelligent programs (Luger 10). The search space can be represented as a graph called a *state space graph* or *search graph* with the nodes representing search states, or objects to be tested (Luger 41; Tyugu 43). A state is connected to another state by an arc that

represents a step in the problem-solving process and paths through the search space represent partial solutions (see fig. 1.3) (Luger 87). State space search by itself is not enough to intelligently solve problems. Algorithms based solely on this tool must search the entire space for a solution, a method known as *exhaustive search* that is impractical for real-world problems. Human problem solving uses *heuristics*, which are "judgemental rules that guide search to those portions of the state space that seem most 'promising'" (Luger 44).

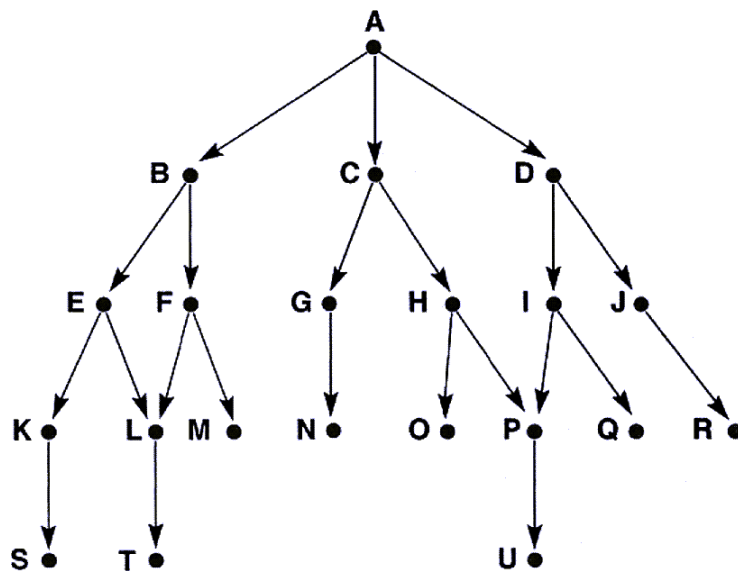


Fig. 1.3: A search graph (Luger 100).

Searches are conducted by starting at an initial state and moving from state to state via arcs. Search algorithms can be characterized by two different measures: time complexity and space complexity. Time complexity is the measure of the amount of time an algorithm takes to complete while space complexity is the measure of the amount of memory an algorithm uses. A related concept, the branching factor, is the

average number of branches, or children, accessible from a particular search state (Luger 158). The number of states at depth d equals the branching factor raised to the d th power. From this, it is possible to estimate the cost of generating a path of a certain length (Luger 158). A large branching factor can have a very bad effect on the time and space complexities of an algorithm and, depending on the problem, render it unsuitable.

Solving a sliding 8-puzzle is an example of a state space search. Though the physical puzzle is solved by moving a numbered tile left, right, up, or down (as long as there is an empty space next to the tile in the intended direction of movement), it is easier to define move rules in terms of moving a blank space as there is only one (Luger 90). If a beginning state and goal state are specified, it is possible to describe the problem-solving process by using search states (Luger 90). This technique is illustrated by figure 1.4.

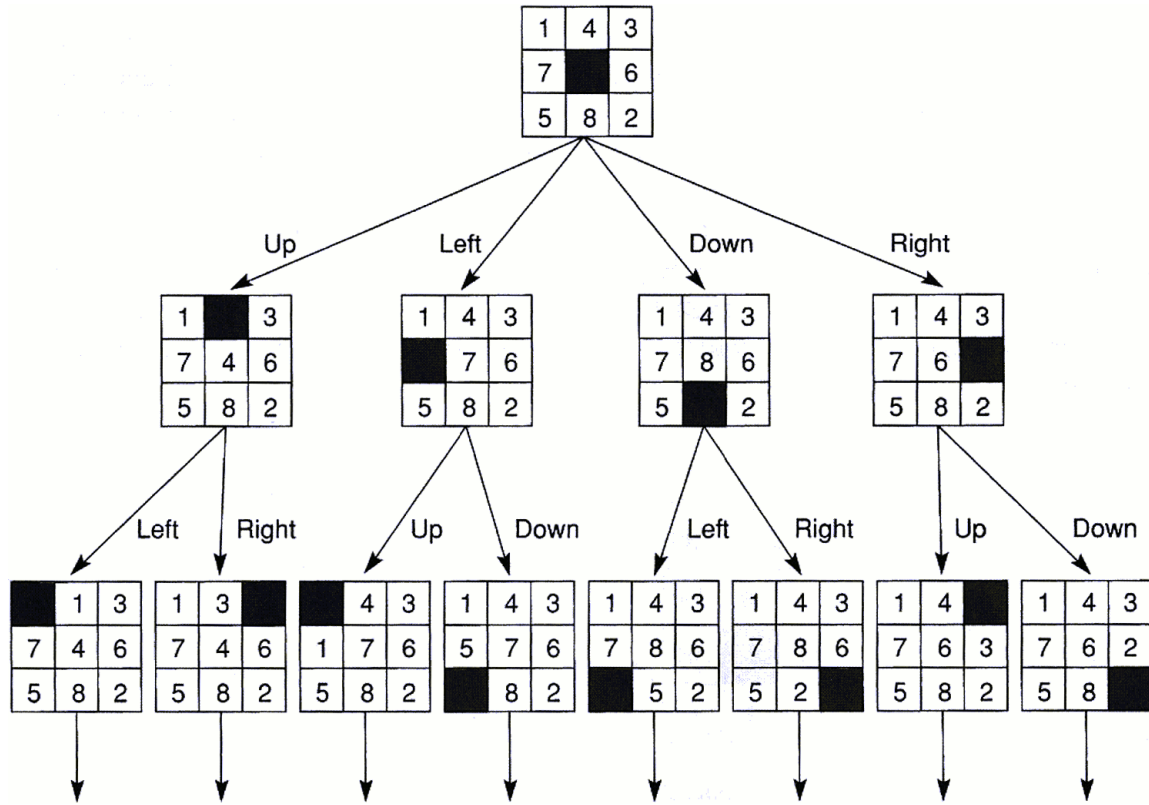


Fig. 1.4. An example of state space search (Luger 90).

There are two types of searches: *data-driven* and *goal-driven* (Luger 93). In data-driven search, the problem solver starts with the facts of the problem and moves toward a solution by adhering to a set of "rules" for changing state. Goal-driven search, on the other hand, starts with the goal and the problem solver identifies a set of rules that can be used to reach it. Both types make use of the same search graph, but the total number of states searched may be different (Luger 94). Choosing one type of search over the other depends on the structure of the problem to be solved (Luger 94).

Exhaustive search algorithms, also known as brute force search algorithms, conduct a systematic search of the entire search space (Luger 44; Tyugu 44). This type

of algorithm is typically used when very little is known about the problem and it is simple enough to be solved without using much intelligence. Two examples of exhaustive algorithms are breadth-first search and depth-first search. With breadth-first search, the search graph is navigated level-by-level, starting from its start state, until a solution is found. Each level is searched completely before the algorithm descends to the next and there is no limit on the number of states or levels in the graph. Figure 1.5 is an example of breadth-first search that searches on the graph of figure 1.3. The states in "closed" have already been examined while the states in "open" are the discovered states whose children have yet to be examined (Luger 99). As all states are first reached from the start state via the shortest path, the path to the solution is also guaranteed to be the shortest (Luger 101). A solution will be found if one exists, which makes breadth-first search "complete" (Chaney, "Breadth-first search"; Choueiry). The search space complexity is proportional to the number of states at the deepest level (Chaney, "Breadth-first search"). Big O Notation, which is defined as $O(x)$ where x is the size of an algorithm's input, is used to characterize the behavior of an algorithm when an input of size x is used. Given branching factor b and depth d , the search space complexity is $O(b^d)$, which may prevent the algorithm from finding the solution given available memory if the path to it is long (Chaney, "Breadth-first search"; Luger 105). The time complexity of this algorithm is also $O(b^d)$ (Chaney, "Breadth-first search").

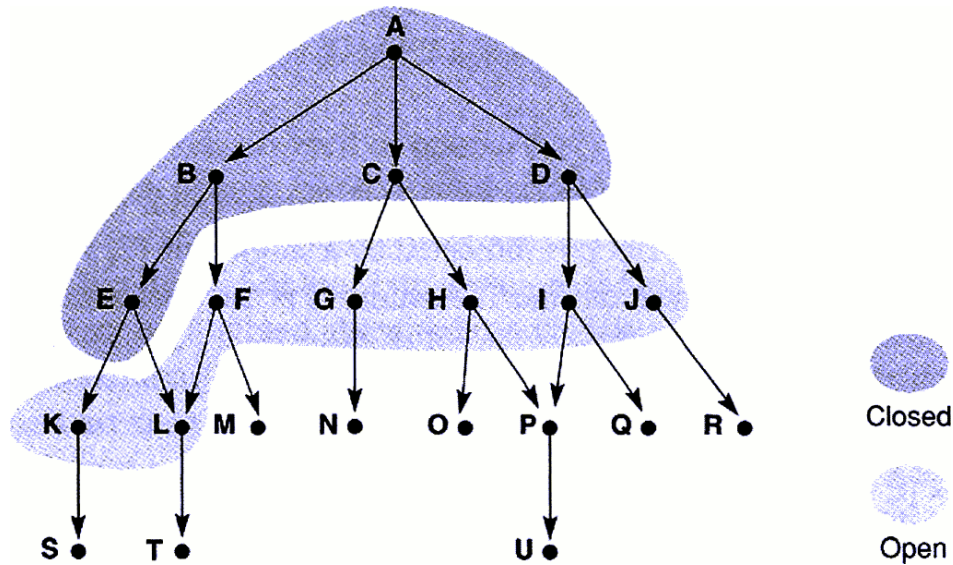


Fig. 1.5. An example of breadth-first search (Luger 101).

A depth-first algorithm searches the graph one branch at a time, beginning at the start state. Each branch is explored as far as possible before the algorithm backtracks to the first untaken branch. Figure 1.6 is an example of depth-first search that uses the same open and closed "categories". It also searches on the graph of figure 1.3. If the search space has many branches, depth-first search is more efficient than breadth-first search (Luger 105). Unlike breadth-first search, depth-first search limits the number of levels to search on a particular branch so that the solution can be found if it is on another branch (Tyugu 47). Also, depth-first search may not find the shortest path to a state the first time it's visited, which indicates that the goal may also not be found using the shortest path (Luger 103-104). Depth-first search has, in the worst case, a space complexity of $O(d)$, which means that its memory usage is dependent upon only the length of the branch, and a time complexity of $O(b^d)$ (Chaney, "Depth-first search"; Choueiry; Luger 106).

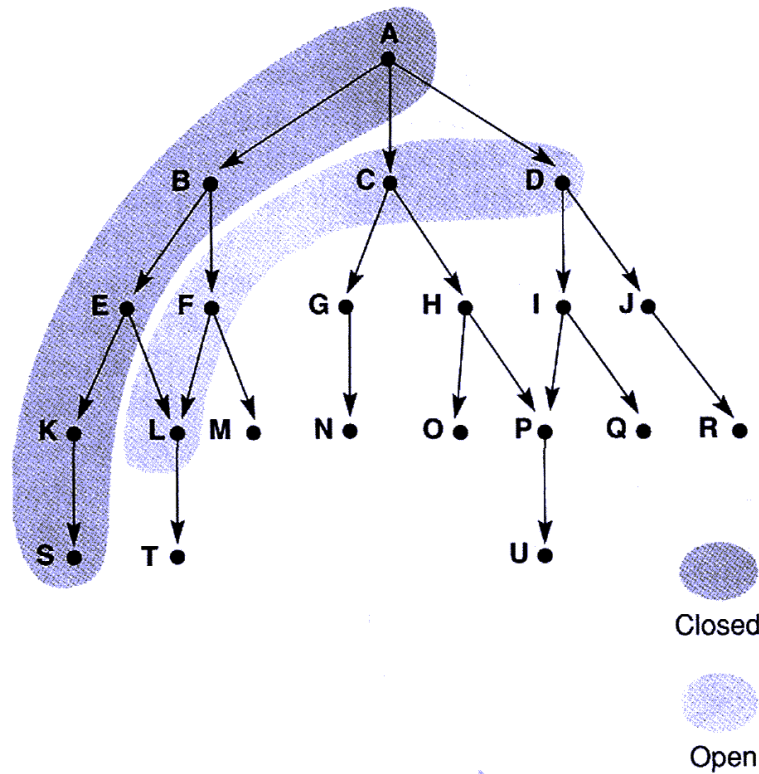


Fig. 1.6. An example of depth-first search (Luger 104).

Unlike exhaustive algorithms, heuristic algorithms use information about the problem to improve the performance of the search (Tyugu 48). They are used when (1) the problem may not have an exact solution, and (2) the problem may have an exact solution but it would take too much computational power or too much time to find it using an exhaustive algorithm (Luger 123-124). Also unlike exhaustive algorithms, a heuristic algorithm may converge to a suboptimal solution or fail to find a solution entirely (Luger 124). Examples of heuristic search algorithms include backtracking, hill-climbing, and best-first.

The backtracking algorithm takes a trial-and-error approach to problem solving (Tyugu 52). A problem is solved by building a solution step-by-step. The search space is

first divided into subspaces. The algorithm selects the first subspace and searches for a suitable candidate solution by using a *fitness function*, which evaluates how well each state, or candidate solution, solves the problem at hand. Each subspace is searched one at a time and a suitable candidate is added to the end of the partial solution. This continues until either a complete solution is found or the partial solution becomes unsuitable (Tyugu 52). If the partial solution must be altered, the last element is dropped and the algorithm backtracks to the subspace where the dropped element was found in order to search again if there are unchecked candidates in the subspace. If there are no other suitable candidates, the algorithm backtracks one more subspace. This continues until either a solution is found or all elements have been dropped from the partial solution and there are no more candidates left to check in the first subspace (Tyugu 52). A variation of this technique can also be used to conduct an exhaustive search (e.g. depth-first) (Luger 99). Figure 1.7 illustrates the backtracking algorithm. The dashed arrows indicate the arcs taken and the direction of movement. For example, one path could be A, B, E, H, backtrack to E, I.

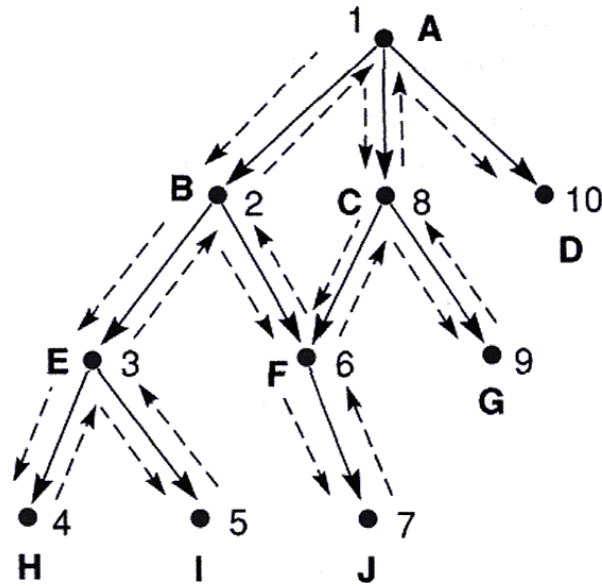


Fig. 1.7. An example of backtracking (Luger 98).

Hill-climbing is the most simple way to implement a heuristic search (Luger 127). Its name originates from a mountain climbing strategy where the climber takes the steepest path uphill until he or she can go no farther (Luger 127). Each child of the current state is evaluated using a fitness function and the best child is selected for further exploration (see fig. 1.8). The parent and sibling states are discarded. Since it only stores the current state in memory, this algorithm is very economical with space (Tyugu 51). However, this also means that it cannot recover if the wrong candidate was selected.

A major problem with hill-climbing is that it may become stuck at local maxima, or states that are better than any of their descendants (Luger 127). Unfortunately, the local maximum may not be the global maximum, which is the best state in the entire search space. "[S]earch methods without backtracking or some other recovery mechanism are unable to distinguish between local and global maxima" (Luger 127).

Hill-climbing can be used to solve traffic flow problems from a source city to a destination city. Using the graph in figure 1.8a, the source city is represented by the letter s while the destination city is represented by the letter t . The arcs between each node represents a street. The goal of traffic flow problems is to send as much traffic from s to t as each street can handle. If path p is generated and only one car is using it as in figure 1.8b, then the residual graph is calculated by subtracting "1" from the capacity of the street (see fig. 1.8c). Path p then has a flow of "1".

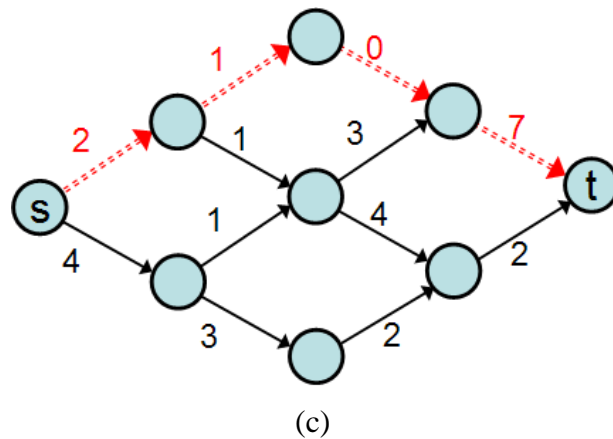
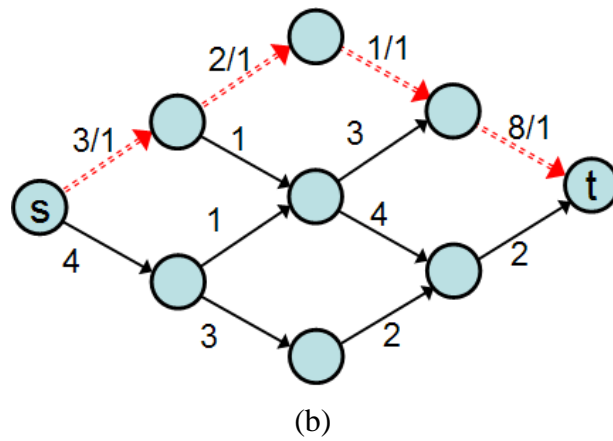
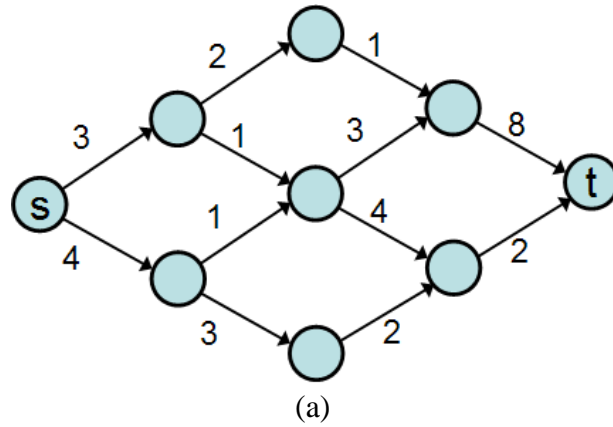


Fig. 1.8. An example of hill-climbing. (a) The source graph. (b) An example of a path from city s to city t . (c) The residual graph. "Algorithms/Hill Climbing"; *Wikibooks, The Free Textbook Project*, 9 Jun 2012; Web; 4 Dec. 2012; http://en.wikibooks.org/wiki/Algorithms/Hill_Climbing

Best-first search is a combination of an improved depth-first algorithm and a breadth-first algorithm (Marshall). The depth-first algorithm is improved by selecting the node one level down that is the best choice instead of simply selecting the first node reached. If none of the descendant nodes are very good choices, the algorithm takes a "breadth-first" approach by examining another node at the same level (Marshall). It also makes use of "open" and "closed" state lists. An example of best-first search is shown in figure 1.9. An example search path is as follows: A-5, B-4, backtrack to C-4, H-3, O-2, backtrack to P-3. The goal of this search is to find the solution by looking at the fewest number of states possible (Luger 135). Unlike hill-climbing, it can recover from taking the wrong path by storing all states previously visited along with those on the same level as the current state that it has not yet evaluated. Storing this information also prevents best-first search from becoming stuck at local maxima (Luger 135-136).

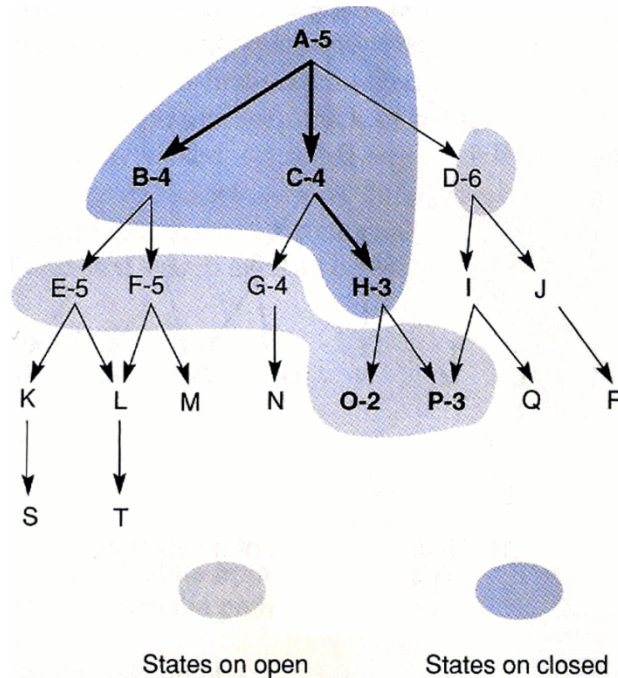


Fig. 1.9. An example of best-first search (Luger 136).

1.3. SEARCH IN AUTO-DESIGN

Artificial intelligence search techniques can be used to automate the design of many different types of systems. The graphical presentation design system, APT (A Presentation Tool), uses a combination of depth-first search and backtracking to systematically explore a search space (Mackinlay). APT describes presentations as sentences of graphical languages that are "compositions of primitive graphical techniques" so that design candidates can be generated and tested systematically (Mackinlay). The algorithm goes through a synthesizing process consisting of three operations before it renders a candidate. Partitioning separates the information to be presented into partitions that satisfy at least one primitive language's criteria for expressing the information with the most important information being partitioned first;

selection selects a primitive language for each partition in which to create a design; composition composes individual designs into a unified presentation (Mackinlay). Backtracking occurs when a partition cannot be matched to a primitive design or when two designs cannot be composed together (Mackinlay).

Another use of search for auto-design is to design configurations for multicore architectures. As the number of cores in a processor rises to tens, hundreds, or even thousands, it becomes impossible to explore all of the possible design configurations. FADSE (Framework for Automatic Design Space Exploration) is a general tool that can be used with various types of search algorithms. It consists of three basic steps. First, a configuration file is read that establishes the search parameters, the objectives, and the constraints used to avoid impossible configurations and to reduce the search space. The data from this file is passed to the Algorithm Runner, which performs the configuration and starts the search process. Next, the jMetal library (which provides computational methods to solve multi-objective optimization problems) generates possible configurations using a search algorithm. These configurations are passed to the Simulator Wrapper, which checks to see if they are valid, then the valid configurations are passed to the Simulation Connector so that each can be evaluated via simulation. The results of the simulation are passed to the Output Reader, which standardizes them before passing them to the Results Receiver. Finally, the results are passed to the jMetal library to be used in the creation of new configuration possibilities and the search process continues.

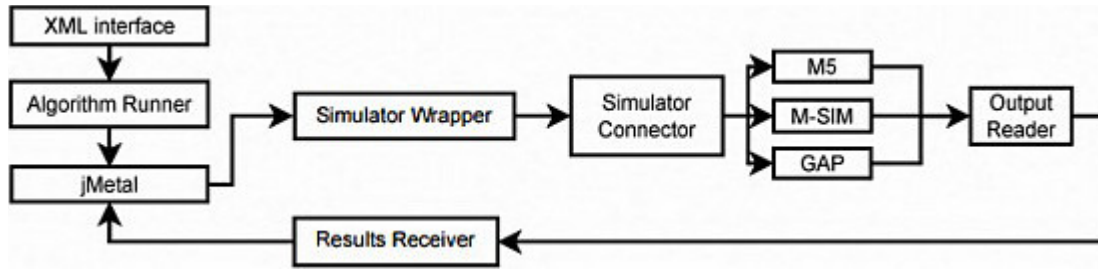


Fig. 1.10. A graphical overview of FADSE (Calborean and Vintan).

1.4. INTRODUCTION TO GENETIC ALGORITHMS

1.4.1. OVERVIEW

A genetic algorithm (GA) is a computational method used in computer science to help develop the best solution to a problem (Jones 115; Koza; Srinivas and Patnaik). Its foundation is rooted in the theory of biological evolution as proposed by Charles Darwin in his famous 1859 book *Origin of Species* (Jones 115; Reeves and Rowe, "Introduction" 2-3). While there are many different programming languages suitable for the development of GAs, the most common is C/C++. A GA consists of four basic processes. The first is initialization, which creates the initial candidate solutions; the second is evaluation, which rates how well each member of the initial group solves the problem at hand; the third is selection, which chooses adequate solutions from the candidate pool and discards the others; and the fourth is recombination, or reproduction, which takes the solutions that survived the selection process and combines them to form new candidate solutions that will hopefully produce a better solution to the problem than the previous generation (Jones 117-119). The final three processes are repeated until a specific end condition is met, which is typically when the algorithm produces a solution that solves the problem. The GA was developed by John Holland in 1962 while the

concept of genetic programming, which is the application of GAs to problem solution, was developed by John Koza in 1992 (Jones 115; Oliver et al.; Reeves and Rowe, "Introduction" 1). Koza wrote that genetic programming is "problem independent," which means that these four basic processes can be used to solve any problem (Koza).

1.4.2. TRADITIONAL GENETIC ALGORITHMS

As the steps of a GA parallel Darwin's Theory of Evolution, each candidate solution is known as a chromosome and each chromosome is comprised of individual variables known as genes that are typically encoded in binary (Jones 115; Koza; Reeves and Rowe, "Introduction" 3; Srinivas and Patnaik). If a problem's variables are not originally encoded in binary then they must be converted (Reeves and Rowe, "Basic Principles" 23 ; Srinivas and Patnaik). In the case of integer variables, it can be as simple as converting each to its binary equivalent (Reeves and Rowe, "Basic Principles" 23). In the case of "real world" variables, such as colors or transportation routes, they are first encoded as integer representations and then typically encoded as a fixed number of binary digits (Srinivas and Patnaik). The process of initialization, which is technically not part of the algorithm, creates a candidate group of chromosomes, typically by using pseudo-random number sequences (Jones 117; Reeves and Rowe, "Basic Principles" 29). Another initialization method is to seed the population with chromosomes that are known to be fit from a previous run of a GA or other algorithm (Jones 117; Reeves and Rowe, "Basic Principles" 29). Regardless of the method, the population must be diverse in order for there to be enough possibilities for adequate solutions, but there is no ideal size (Reeves and Rowe, "Basic Principles" 25). Empirical results from several authors

have suggested that a population of 30 chromosomes is "quite adequate in many cases" (Reeves and Rowe, "Basic Principles" 25). D. E. Goldberg found that basing population size on chromosome length is also adequate: the longer the chromosome, the larger the population (Reeves and Rowe, "Basic Principles" 25). Initialization is usually performed only once per problem.

After initialization, each chromosome in the population is evaluated by how well it solves the given problem (Jones 118; Reeves and Rowe, "Basic Principles"). This process almost always produces a single value known as a fitness rating for each chromosome (Koza). Large values are better than small values (Jones 118). The criteria, or fitness measure, used to evaluate each chromosome is typically set by the programmer and depends on the problem to be solved (Koza). There can be a range of acceptable values or only one (Koza).

The third process, selection, separates the chromosomes that solve the problem at hand from those that do not. This process parallels Darwin's *survival-of-the-fittest* theory: the chromosomes with high fitness ratings will be selected for breeding while the ones with low ratings will be discarded (Srinivas and Patnaik). Jones maintains that selection is "quite possibly the most important and most misunderstood step of [a GA]" (118). If the selection is too specific, the population will not be diverse enough to create a better solution due to lack of possibilities; however, if the selection is too broad, the fitness of future solutions will not improve (Jones 118).

There are numerous algorithms available to separate the fit chromosomes from the unfit (Jones 118). In the original scheme, *roulette-wheel selection*, a string's probability of being selected is proportional to its fitness (Bauerly and Liu; Jones 118-

119; Reeves and Rowe, "Basic Principles" 31; Srinivas and Patnaik). If depicted graphically, these wheels would look very similar to a pie chart. A single pointer is typically used to make each selection, which means that the wheel must be "spun" at least once for every chromosome. The pointer represents a number generated by a random number generator (Reeves and Rowe, "Basic Principles" 31). Figure 1.11a is an example of single pointer selection. If the value 0.13 was randomly generated, the chromosome selected would be number 1. There is an alternate scheme based on roulette-wheel selection called *stochastic universal selection* that uses a pointer for each chromosome spaced equally around the wheel, which means that only a single spin is necessary as all chromosome selections are made simultaneously (see fig. 1.11b) (Reeves and Rowe, "Basic Principles" 31-32). Although the stochastic universal selection scheme has been proven to produce results that are a very similar to natural selection, many published works use the traditional method (Reeves and Rowe, "Basic Principles" 32). One of the problems associated with roulette-wheel selection is that how well it works depends on the size of the population: large populations generate better results than small ones (Srinivas and Patnaik).



Fig. 1.11. A visualization of roulette-wheel selection. (a) Single pointer selection. (b) Stochastic universal selection. (Reeves and Rowe, "Basic Principles" 32)

Another popular selection scheme is *tournament selection* (Jones 144; Reeves and Rowe, "Basic Principles" 34-36). In strict tournament selection, two chromosomes are compared against each other and the one with the higher fitness rank is allowed to reproduce (Jones 144). Tournament selection is completed twice in order to select two parents (Jones 144). A fourth scheme known as *proportionate selection* allocates a set number of children to a chromosome based on its fitness rank divided by the average rank of the entire population (Srinivas and Patnaik). Therefore, chromosomes with a higher than average fitness rank are allocated more than one child and those with a lower than average rank are allocated a fractional child (Srinivas and Patnaik).

The chromosomes deemed most fit will be allowed to move on to the next process, which is known as recombination (Jones 119; Reeves and Rowe, "Basic Principles" 38). New chromosomes are created by using one or more methods (Jones 119). The most common genetic operators used for recombination are crossover and mutation (Bauerly and Liu; Cho; Jones 120; Srinivas and Patnaik). It is common to use both, but some researchers have discovered that only one was necessary in certain cases (Reeves and Rowe, "Basic Principles" 30-31).

In crossover, two parent chromosomes give genes to create two child chromosomes (Jones 120; Reeves and Rowe, "Basic Principles" 38). Each parent's genes are randomly selected and then combined to form a new chromosome (Jones 120; Reeves and Rowe, "Basic Principles" 38; Srinivas and Patnaik). There are two types of crossovers: *single-point* and *multi-point* (Jones 120; Reeves and Rowe, "Basic Principles" 38; Srinivas and Patnaik). Let l represent the length of a chromosome. With single-point crossover, a number is chosen at random from the range 1 to $l-1$ to serve as the *crossover point* (Jones 120; Reeves and Rowe, "Basic Principles" 38; Srinivas and Patnaik). The group of genes before and after this point are swapped in the parents to create two children (Jones 120; Reeves and Rowe, "Basic Principles" 38; Srinivas and Patnaik).

For example, Parent A and Parent B both have genes numbered from 1 to l . Both have a length of 4. If the crossover point was 2, Child A would receive genes {a1, a2, b3, b4} and Child B would receive genes {b1, b2, a3, a4} (see fig. 1.12a). Multi-point crossover, in contrast, results in two or more randomly-selected crossover points and three or more groups of genes in each child (see fig. 1.12b) (Jones, fig. 6.8; Reeves and Rowe, "Basic Principles" 38; Srinivas and Patnaik). It is not required for crossover to occur even if the programmer chooses to use it (Reeves and Rowe, "Basic Principles" 43; Srinivas and Patnaik). A crossover occurs only if a randomly generated number in the range from 0 to 1 is greater than what is known as the *crossover rate* (Srinivas and Patnaik). In a large population, this number is the fraction of chromosomes that have been crossed (Srinivas and Patnaik).

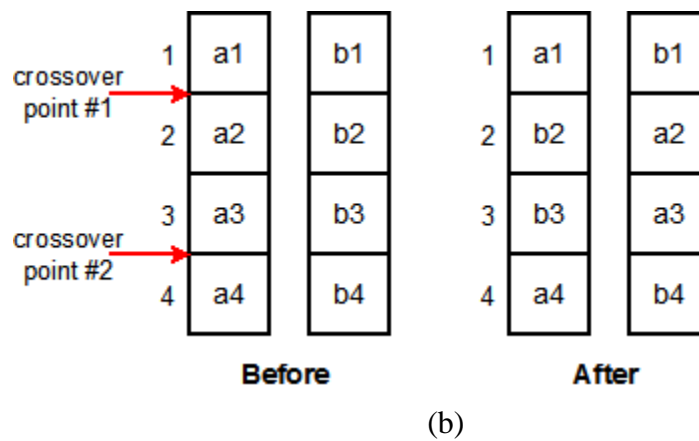
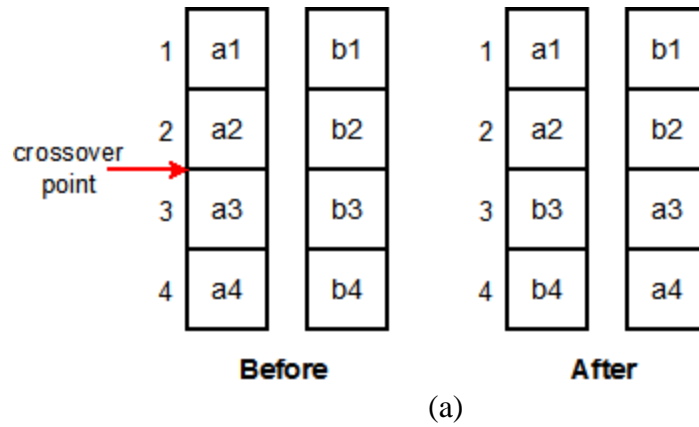


Fig. 1.12. An example of chromosome crossover. (a) Single-point. (b) Multi-point.

When the mutation operator is used, a gene is randomly selected and changed (see fig. 1.13). This operator can be used to introduce new material into the solution space (Jones 120; Reeves and Rowe, "Basic Principles" 44; Srinivas and Patnaik). This is very useful when a particular gene has the same value in every chromosome (Srinivas and Patnaik). The mutation process is controlled by a *mutation rate*, which determines the odds of a gene being changed (Srinivas and Patnaik). If mutation is looked upon as an option and not a necessity, then a decision must be made as to whether or not it

should be used for a particular problem and at what rate (Reeves and Rowe, "Basic Principles" 44).

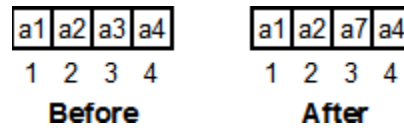


Fig. 1.13. An example of chromosome mutation.

The parents remain in the candidate pool with the children after the recombination process completes and the algorithm restarts at the evaluation process. The total number of times a GA executes is determined by a predefined ending condition. Common ending conditions include stopping when a certain number of fitness evaluations have been performed and stopping after a specific amount of time has elapsed (Reeves and Rowe, "Basic Principles" 30).

A GA can be adjusted to produce better results (Jones 144). For example, an alternate selection scheme or a different combination of genetic operators can be used. No scheme or operator will work for every problem (Jones 144-145). One simple way to modify parameters is to alter the size of the candidate population. Large populations typically work better than small ones. The population size is also linked to the selection of the crossover and mutation rates (Jones 145).

1.4.3. INTERACTIVE GENETIC ALGORITHMS

An interactive genetic algorithm (IGA) differs from a traditional genetic algorithm during the evaluation process. The user decides the fitness of the candidate solutions instead of the algorithm. Interactive genetic algorithms are used when a solution cannot be found mathematically, such as when what is considered a good solution is entirely subjective (Monmarché et al.). Users must be able to easily evaluate candidates, which implies that they must be visualized (Monmarché et al.). IGAs can present possibilities to the user that he or she may not have considered (Monmarché et al.).

1.4.4. USING A GENETIC ALGORITHM FOR AUTO-DESIGN

A GA can be used to solve one of the most famous problems in AI, the Traveling Salesman (or Salesperson) Problem (TSP) (Luger 513; Reeves and Rowe, "Introduction" 6). The problem statement is as follows:

A salesperson is required to visit N cities as part of a sales route. There is a cost (e.g. mileage, airfare) associated with each pair of cities on the route. Find the least cost path for the salesperson to start at one city, visit all the other cities exactly once and return home. (Luger 513)

The fitness function used for this problem is simple: evaluate the cost of the path. The genetic representation of this problem, however, is much more complex. For example, if there were four cities on the route and each city was encoded as a four bit integer (0001, 0010, 0011, 0101), then it would not be possible to use either crossover or mutation (Luger 514). Crossover could remove a city from the route, include a city more

than once, or even introduce an invalid city (e.g. performing crossover on 0001 and 0010 at position 2 would produce children 0000 and 0011). Mutation could also introduce an invalid city (e.g. mutating 0001 at position 0 creates 1001, or 9). There are other methods of encoding that could be used to prevent these problems (such as using integers instead of bits) as well as other genetic operators such as *order crossover*, which preserves the order of the cities in the partial solutions of the parents. A portion of parent #1's bit string is inherited by child #1 while the remainder of the string and the order of its elements is inherited from parent #2 and vice-versa (Luger 515). Figure 1.14 is an example of TSP.

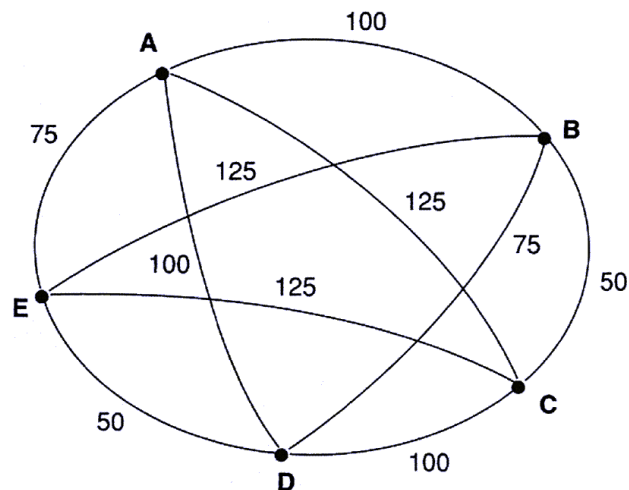


Fig. 1.14. An example of the Traveling Salesman Problem (Luger 91). The cities are lettered and the travel cost between cities appears on each arc.

Non-cryptographic hash functions, which are used to index records in databases or match everyday website addresses to numerical IP addresses, have also been developed using a GA. The function is assembled by combining various smaller base functions and is evaluated by measuring the total number of collisions that occurred

while hashing 10^6 32-bit random strings (Karásek). A collision occurs when two or more nonidentical strings evaluate to the same hash value when using the same function. Collisions are not entirely avoidable, but a good hash function keeps the number of collisions as low as possible.

IGAs have been applied to the development of web page designs. A team of researchers at the Université de Tours in France developed an algorithm that could be used to create a web page by someone who knew nothing about web design (Monmarché et al.). Initially, only the manipulation of Cascading Style Sheets (CSS) was explored, which are used to add visual styling to web pages (Monmarché et al.). Later, when the manipulation of HTML was attempted, the underlying structure of the web page was generated by the IGA (Monmarché et al.). In both cases, the algorithm guided the user through the creation of a web page by making suggestions that the user would then select from (Monmarché et al.). Each generation of candidates was displayed to the user, 12 to a page, so that all candidates were visible on the user's monitor at the same time. Once an initial selection had been made from a general group of candidates, the algorithm then narrowed down the options by using subsequent selections until the user made a final decision or there was only one candidate remaining (Monmarché et al.). User selections guide the evolution of the candidate population (Monmarché et al.).

The original *Imagine* tool used 26 visual attributes as genes with values that the user could change according to personal preference (Monmarché et al.). It also used a concept that the creators called *gene frequencies* to create a diverse initial candidate pool of a dynamic size while still limiting the population so that every candidate in the pool could be clearly displayed to the user simultaneously (Monmarché et al.). Gene

frequencies utilizes a vector of probabilities to represent an infinite population and chromosomes are generated according to these probabilities (Monmarché et al.).

An IGA has also been used for fashion design (Cho). As with web page design, a "good" fashion design cannot be decided using a traditional genetic algorithm as the decision is entirely subjective. The fashion design system has two main parts: an IGA and an OpenGL program that generates a 3D model for each chromosome. A separate algorithm is used to decode the bits of each chromosome into a format usable by the OpenGL program. The candidates are then displayed to the user, eight at a time.

CHAPTER TWO: A WEB APPLICATION TO GENERATE WEB PAGE DESIGNS

2.1. APPROACH

2.1.1. OVERVIEW

The IGA is a PHP web application (web app) that consists of three main steps: initialization, evaluation, and recombination. The initialization step creates candidates based upon default or user-selected initial settings. The evaluation step displays screenshots of the candidates to the user as enlargeable thumbnail images. The recombination step, which is the most important, generates new candidates to replace those not selected by the user via crossover and mutation operators at fixed rates. If neither of these operators are used, the IGA generates a candidate from "scratch" via initialization instead.

The general outline of the web app is as follows:

1. Give the user a chance to change initial settings such as the web page layout, the colors to be used, and the font.
2. Randomly generate a candidate population.
3. Allow the user to evaluate candidates.
4. Perform recombination on the current population.
5. Repeat steps 3 and 4 until the user settles on a single candidate, decides to start over, or quits the web app.

Figure 2.1 provides a visual overview of the IGA process.

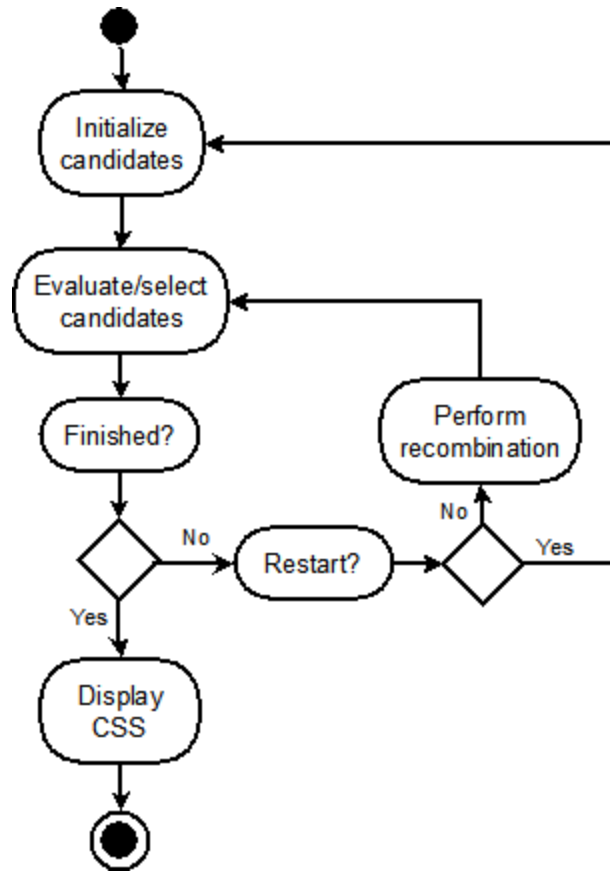


Fig. 2.1. An overview of the IGA process.

The web app evolves style sheets used to define the visual appearance of a web page that is mostly static. Cascading Style Sheets (CSS) is a scripting language consisting of selectors that indicate which portions of a web page to style and properties that modify the attributes of selectors. The web app dynamically creates a style sheet for each candidate.

Early explorations by Monmarché, et. al. with their *Imagine* tool used an HTML table to define the page layout in addition to a style sheet; however, this is not an accepted best practice in website development today (Raggett et al.). HTML was originally developed to define the basic structure of a web page while CSS was

developed to define a web page's appearance (Raggett et al.). The web app described by this thesis does not evolve HTML at all.

The web application was created using PHP 5.3 in a Microsoft Windows 7 environment using the network loopback interface, which enables a single PC to act as both the client and the server. The web server used for development was Abyss Web Server X1 version 2.8.

While there is no guarantee that the IGA will produce the optimal solution, the size of the search space will allow it to present a large variety of different designs to the user. There is a total of 20 CSS selectors (ID selectors as well as HTML element selectors such as "body" and "p") (see fig. 2.2) and a total of 27 CSS properties (see fig. 2.3) in use that can be divided into two categories: visual and text. Not all properties are applicable to every selector, however. For example, only properties that alter visual characteristics are used for images. Unrestricted, the search space is a staggering 1.6×10^{343} style sheets. However, to guarantee specific layouts, some properties are restricted and their value either depends upon what was selected in the initialization settings or upon the value generated for a previous property. The size of the search space, when taking into account restrictions, ranges from 5.3×10^{81} to 7.8×10^{313} style sheets. Even though it is always very large, it is still possible for the web app to generate satisfactory designs because it takes into account the user's preferences.

<body>	Web page body.
#container	Web page container.
#header	Web page header section.
#nav	Web page navigation section.
#sidebar	Web page sidebar section.
#content	Web page content section.
#footer	Web page footer section.
<h1>	1st level heading.
<h2>	2nd level heading.
<h3>	3rd level heading.
<p>	Paragraph.
<hr>	Horizontal rule (section divider).
	Unordered (bulleted) list.
	List item.
	Image.
<a>	Anchor (link).
:link	Unclicked link psuedoselector.
:visited	Visited link psuedoselector.
:hover	Mouse pointer hover link psuedoselector.
:active	Mouse click link psuedoselector.

Fig. 2.2. Page element selectors.

Visual Layout Properties	
background-color	Element background color: transparent, inherit, hexadecimal.
width	Element width: auto, inherit, percentage.
height	Element height: auto, inherit, pixels.
margin-top	Element top margin: auto, inherit, pixels.
margin-bottom	Element bottom margin: auto, inherit, pixels.
margin-left	Element left margin: auto, inherit, pixels.
margin-right	Element right margin: auto, inherit, pixels.
padding-top	Element top padding: inherit, pixels.
padding-bottom	Element bottom padding: inherit, pixels.
padding-left	Element left padding: inherit, pixels.
padding-right	Element right padding: inherit, pixels.
line-height	Element line height: normal, inherit, pixels, percentage.
display	Element display: block, inline, none.
float	Element float positioning: none, left, right.
clear	Element float clearing: left, right, both.
border-width	Element border width: inherit, pixels.
border-style	Element border style: solid, dashed, double, dotted, inherit.
border color	Element border color: inherit, hexadecimal.
Text Layout Properties	
font-family	Element font face: serif, sans-serif, monospace, inherit
font-size	Element font size: inherit, pixels, percentage
font-weight	Element font weight: normal, bold, inherit
font-style	Element font style: normal, italic, inherit
font-variant	Element font variant: normal, small-caps, inherit
color	Element font color: inherit, hexadecimal
text-transform	Element text transformation: none, uppercase, lowercase, capitalize, inherit
letter-spacing	Element letter spacing: inherit, pixels
text-align	Element text alignment: left, right, center, justify, inherit

Fig. 2.3. CSS properties with valid values/types.

2.1.2. THE ENCODING MECHANISM

Depending on the property, a valid value can either be a number (integer or hexadecimal) or a string. A numeric value is easily encoded using its binary equivalent, but a string value must first be represented as an integer. A list of possible string values for a particular property are numbered sequentially starting from zero. The values are then encoded using a fixed number of binary digits, the total length depending upon the value of the largest integer to be encoded. The fewest number of bits necessary to encode the largest integer value is the rule for all properties. The property's value becomes the first portion of the property's section of the bit string. The second portion, the two bit value type, is what differentiates two otherwise equivalent numerical values of two different property types. It distinguishes 10 pixels from 10 percent, for example. The four value types are: pixel, percentage, hexadecimal, and string.

As an example, the property *font-size* can use values belonging to several different types. I make use of three of them: pixel, percentage, and string. The range of valid values for pixel is 12-25 while the range of values for percentage is 80-125. The only valid string value is *inherit*, which is mapped to an integer value of 0. The largest value to encode is 125, which requires seven bits minimum, so this is the length of the value portion of the bit string for the *font-size* property.

The bits representing the properties of a selector are concatenated together in a specific order (see fig. 2.4c). If a property does not apply to the selector, it does not appear in the bit string. The selectors (and their associated properties) are also concatenated together to form the chromosome as a whole (see fig. 2.4a). Every chromosome has the same number of selectors.

- (a) {[properties of <body>][properties of #container] ... [properties of :active]}
- (b) {[bits 0-134][bits 135-273] ... [bits 3543-3706]}
- (c) [(background-color)(margin-top) ... (text-align)]
- (d) [(bits 0-25)(bits 26-31) ... (bits 130-134)]

Fig. 2.4. Bit string organization for a chromosome. (a) Organization of selected CSS selectors. (b) Bit ranges of selected CSS selectors. (c) Organization of selected properties of <body>. (d) Bit ranges of selected properties of <body>.

2.1.3. WHY PHP?

PHP was originally selected simply because it was the web programming language that I was most familiar with. I wanted the IGA to be a web application so that others could use it to generate their own designs. As I was coding, I realized that PHP has an important feature that makes it well-suited for my application.

Since PHP arrays are actually ordered maps, the indices are not limited to integers. I use strings as array indices numerous times within the application. Another consequence is that numerical indices can be skipped. There are also many useful array functions available such as *array_count_values* which returns the frequencies of the values in an array.

There is one significant limitation when using PHP, however. The only built-in method to create screenshots is the PHP COM extension. COM, or Component Object Model, is a Microsoft technology and only works in a Windows environment. You are also required to use Internet Explorer (IE) to open web pages. A screenshot of each candidate is taken after the style sheet link in the generic static web page has been

updated to point to the current candidate's style sheet. The screenshot creation process is very slow, especially when using a browser other than IE to run the web app. When creating a population of 20 candidates during the initialization stage, it takes over two and a half minutes to finish the process when using Firefox. When using IE, the total time taken is about a minute less, which is due to the fact that the program is already running when the static web page is loaded into a new window. The maximum allowable run time for PHP scripts must be extended so that initialization does not time out when creating the original candidate population. It was also necessary to extend the maximum allowable script run time of the Abyss Web Server.

2.2. THE ALGORITHM

2.2.1. INITIALIZATION

Initialization begins with a configuration page (see fig. 2.5). There are five categories of settings: the overall page layout; the main color scheme; visual layout properties such as width and height; border; and text-related properties. The value or range of values selected for a property applies to every selector on the page that uses the property. While the user has access to the full range of valid values for each property available on the page, not all properties can be modified by the user. Some properties, such as *display* and *float*, are used to define the page layout and must have a specific value or a value within a specific range in order to achieve certain layouts. There is a special value, *random*, that allows the algorithm to choose any valid value for a property regardless of type.

Interactive Genetic Algorithm for Website Design

Initial Settings

This app is meant to assist with website design ideas. **DISCLAIMER:** I am making absolutely no guarantee that it will create something good on the first try or even on the 100th. Also, the design may not look the same in all browsers (most notably any version of IE older than 9). Use for entertainment purposes only.


NOTE: When "inherit" is selected a webpage element (<body>, <p>, etc.) will use the value of the closest ancestral element with a value for the property that is something other than "inherit". In other words, the value for that property will be inherited. If there is no value other than inherit, the default browser value will be used. For example, if you choose "inherit" for letter-spacing and no previous element has a value other than "inherit" for letter-spacing then it will use the browser default, which is "0px".

TIP: If you want variety in the design, either choose a wide value range or choose "random".

There are a total of three stages in the interactive genetic algorithm: (1) Initialization, (2) Evaluation, (3) Recombination (optional). Change as few or as many of the initial settings as you want. [Click here](#) for more detailed instructions and background information.

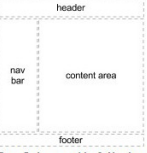
Page Layout

Choose one of the five layouts or let the app choose for you by selecting "random". **NOTE:** The columns in the two-column layout with the sidebar and the three-column layout may not be in the exact order as the example. The first one has 2 layout possibilities while the second has 4.



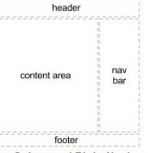
One-Column w/ Top Navbar

☒



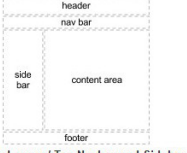
Two-Column w/ Left Navbar

☐



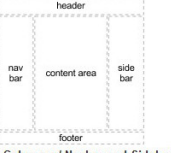
Two-Column w/ Right Navbar

☐



Two-Column w/ Top Navbar and Sidebar

☐



Three-Column w/ Navbar and Sidebar

☐

Random Layout

☐

Fig. 2.5. The IGA configuration page.

After the configuration form is submitted, a cookie is created that stores a randomly generated session ID used to link the client to a server session. A new PHP session is then established so that certain variables, such as the initialization settings and the candidate pool, will be accessible from all scripts. Next, the data passed from the initialization form to the PHP initialization script is parsed and stored within an array that uses strings corresponding to CSS properties as indices. This array is also stored within the session array variable associated with the current session.

Next, the script builds the bit string for each candidate. The total number of candidates created depends on the the user's screen resolution. If the screen is less than 768 pixels in height, the script generates 15 candidates; otherwise, it generates 20 candidates. For each applicable property, it generates a value by using both the

initialization settings and the page layout settings as guidelines. If random value selection is set for the property, the script first randomly selects from amongst the property's valid value types before randomly selecting a value from a range of valid values for that type and converting it to binary. Each newly-converted value and its type is appended to a temporary bit string. Once all of the property values are generated for a selector, the temporary bit string is then appended to the overall bit string for the current candidate. The CSS is constructed at the same time as the bit string. The original value generated by the initialization script is the value that is used for the CSS. Each value and value type is stored in a two-dimensional array that uses the numerical IDs of the selectors and properties as indices.

Colors are handled in a slightly different way from integer and string values. The list of hexadecimal color values that have been previously designated as the overall color scheme is converted to its binary equivalent. This process also occurs for the list of text colors if the user has not chosen to use the main color scheme. When a color value is necessary, a color is randomly selected from the appropriate list and appended to the temporary bit string along with two bits that represent the hexadecimal type.

Once values have been generated for all of the properties applicable to all of the selectors, a new candidate object is created. The bit string is stored within the candidate along with the CSS array. A CSS file is created using the CSS array along with a screenshot and a much smaller thumbnail image of the candidate. The file paths of the style sheet file, the screenshot, and the thumbnail are also stored within the candidate object. The object is then stored in an array that represents the candidate pool along with the rest of the candidate objects.

2.2.2. EVALUATION

The user evaluation step is simple. A PHP session is started using the session cookie set during initialization. A table is created to display all candidates in the candidate pool using small versions of the design screenshots (thumbnails). Below each thumbnail is a checkbox used to select the candidate if the user wants it to survive the current generation and possibly be used to create new candidates for the next generation. Once the user selects "recombine", the list of selected candidates is then passed to the recombination script. When the evaluation step is repeated, only the candidates that were selected during the most recent round of evaluation and the new candidates created during recombination will be displayed to the user.

2.2.3. RECOMBINATION

During recombination, new candidates are created to replace those the user did not select. The data passed from the evaluation page is processed. If the user clicked "finish", the candidate that he or she selected as their final choice is then displayed. If the user clicked "restart", the initialization settings page is displayed. Otherwise, the recombination process begins.

It can be described as follows:

1. Record the fitness of each candidate.
2. Check the number of selected candidates.
 - a. If one candidate is selected, repeatedly perform mutation on it until the current population size equals the maximum size n of the candidate pool.
 - b. If between two and $n-1$ candidates are selected, generate a probability distribution that assigns a selection probability to each candidate that is proportionate to its fitness. Perform multipoint crossover and/or mutation on candidates selected via roulette-wheel selection. Possibly perform mutation on children created via crossover. As a last resort, generate a brand new candidate. Do this until the current population size equals n .
 - c. If all candidates are selected, propagate the entire generation.
 - d. If no candidates are selected, recreate the entire generation.

After each round of user evaluation, the number of times a candidate has survived evaluation is increased by one and the value for the candidates that did not survive is reset to zero. This "survival value" is also equal to the fitness of each candidate. The fitness determines the probability of a candidate being selected for crossover.

The probability distribution is created by dividing each candidate's fitness by the sum of all the fitness values of the candidates selected by the user. Ideally, the roulette values of all selected candidates added together would equal "1", but it is sometimes necessary to adjust the values to make this true. If the sum is not equal to "1", the frequency of the fitness values is calculated and the result is sorted in reverse numerical order by array index. If there is a fitness value with a frequency of "1", the ID of the candidate is identified and its fitness value gains the unallocated remainder of the distribution value ($1-\text{sum}$). If no fitness value appears only once, the maximum value of the distribution is defined as the unadjusted value and a small margin of error is

introduced. This distribution sets the values of the roulette wheel used to select parents for crossover.

If recombine was selected on the evaluation page, new candidates are created through the use of the crossover and/or mutation operators at a rate of 0.6 and 0.05 respectively, or through initialization. If only one candidate was selected by the user during evaluation, it is possibly mutated to create a new candidate by comparing a randomly generated floating point number to the mutation probability p_M . If the random number is greater than p_M , the new candidate is initialized instead.

If a candidate was selected for mutation, a bit in its bit string is randomly selected and then subsequently flipped (a "1" becomes a "0" and a "0" becomes a "1"). If the bit that was flipped was part of a property type portion of the string instead of a value portion, it is sometimes necessary for the value to change to match the new type and its associated value restrictions. For example, the value of *#header margin-top* before mutation was 10 (1010) and of type pixel (01), but the value afterward became type string (00) and the maximum defined value for that type for *margin-top* is 1 (0001), or "inherit". This is shown in figure 2.6. The CSS value associated with this selector and property is also updated to reflect the change. A new candidate object is created based on the updated bit string, a new CSS file is created based on the updated CSS array, and a screenshot is taken before the candidate is inserted into the candidate pool to replace a candidate that the user did not like.

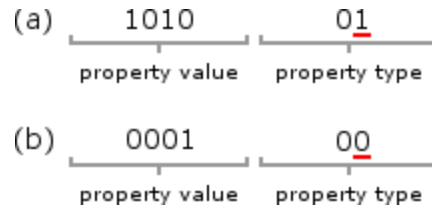


Fig. 2.6. Example of property type change caused by mutation. (a) Before mutation. (b) After mutation and correction. The flipped bit is underlined.

It is possible for undefined genetic material to be introduced into the candidate population. If a property's value is mutated to something outside the range of previously defined values, and the value's type is "pixel", "percentage", or "hexadecimal", it is allowed since the CSS 2.1 specification considers these values to be acceptable. If its type is "string", a defined value must be randomly selected to replace the undefined mutated value. String values are mapped to integers and unmapped values are meaningless. By allowing properties to take on values previously undefined, users are given the chance to explore more of what CSS has to offer.

Certain properties of certain selectors are not allowed to be mutated in order to preserve the page layout. *#header width* is one example. If a bit belonging to one of the forbidden properties was flipped, the mutation is reversed, the CSS update is discarded, and the mutation function returns as if the candidate was never selected for mutation.

If the number of candidates selected is between two and $n-1$, two candidates are chosen using roulette-wheel selection and possibly mated. A floating point number is randomly generated and then compared to each value on the roulette wheel until a candidate is selected. If the random number is less than or equal to the roulette value of the candidate currently being tested, the previous roulette value is then compared to the

random value. If it is greater than this number, the current candidate is selected. This process is then repeated for the selection of the second parent. If the first parent is selected a second time, the process is repeated until both parents are unique.

The crossover process clones two parent candidates, randomly chooses two points in each bit string, and swaps the bits between these points so that two new child candidates are created. If the length of the bit string is *len*, the selection of the first point can be made from any position in the range of 0 to *len-1* (inclusive). The selection of the second point, however, is much more restricted. The minimum value of the range is the position offset of the selector associated with the first point while the maximum value is the minimum value + one less than the length of the selector's portion of the bit string. The selection of the second point must be carefully controlled so that all bit string swaps occur within the boundaries of a single element selector. Otherwise, swapping the values of the two-dimensional CSS array if the portion to be swapped overlaps two or more selectors will be inefficient. Figure 2.7a shows two crossover points that fall within the boundaries of both the *<body>* (bits 0-134) and *#container* (bits 135-273) selectors while figure 2.7b shows the points after they have been adjusted to fall within the boundary of only the *<body>* selector.

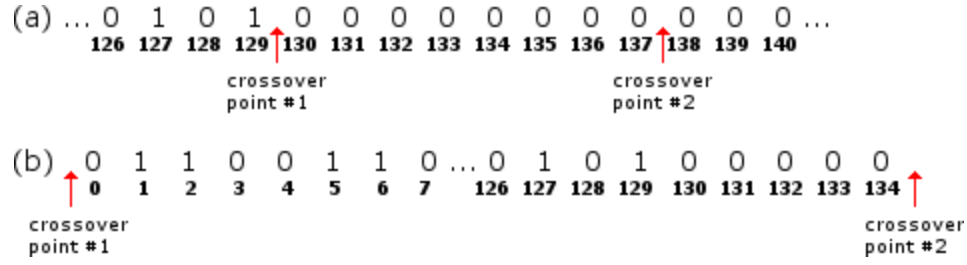


Fig. 2.7. An example of crossover point selection. (a) Before adjustment. (b) After adjustment.

If the two crossover points are identical, the second point is selected again until it is unique. Afterward, the points are sorted in ascending numerical order, which means that the first and second points may swap places. The numerical IDs of the properties that each point is associated with are identified and used to adjust the points to match the start and end positions of the properties. For example, if point #1 is located in the portion of the bit string associated with *#header width* while point #2 is located within the portion associated with *#header margin-top*, point #1 is adjusted to be the first bit of *#header width* while point #2 is adjusted to be the last bit of *#header margin-top*. This prevents partial property values from being swapped. Finally, the values corresponding to the selector and properties in the previously cloned CSS arrays are also swapped.

The two new children are initialized with the altered bit strings, CSS files are created from the altered CSS arrays, and screenshots are taken. Each child is then possibly mutated in an attempt to introduce greater variety into the candidate population.

If the randomly generated floating point number is greater than the crossover probability p_c , a candidate is randomly selected for possible mutation instead. If mutation fails, then a brand new candidate is initialized. Testing for crossover continues

until either all available positions are assigned to new candidates or the number of available positions equals one. If there is only one position remaining, a new candidate is created using either mutation or initialization.

If every candidate was acceptable to the user, the current generation is propagated to the next with no alteration. If no candidates were acceptable, the entire population is discarded and another is created via initialization.

2.2.4. FINISHING

Once the user has settled on a final design by selecting "finish" on the evaluation page, a simple "finishing" script is executed. The user is unable to undo this action once completed. If he or she wishes to explore more design possibilities, he/she has no choice but to start over from the beginning.

2.3. THE USER INTERFACE


The web app has three user interfaces. The initialization settings page is the most complex. It consists of five categories of settings that may be altered by the user or left as the default. Depending on the property, it is possible to choose a single value, a range of acceptable values, or allow the web app to randomly choose a value from a predefined list of acceptable values. The user is allowed a little more freedom of choice in terms of colors. A selection can be made from one of four predefined main color schemes or he or she can input between two and six colors of his or her own. For text color, the user can input between one and three if he or she does not want to use the


colors of the main scheme. One of five general web page layouts can also be selected or the user can allow the web app to choose.


Main Colors


Choose from a color family or input up to six of your own color values in hexadecimal. Using your own will override any selection you make in the color family section.

Color Families

☒ 

☐ 

☐ 

☐ 

☐ Random

Use Your Own

Choose between two and six colors (three to five is best). You can find a color palette [here](#). The hexadecimal value is either on the right beside the color or on the left underneath the palette. **NOTE:** You must use two or more *consecutive* text boxes.

(a)

Text Properties

These settings are related to text display. Choosing a radio button not associated with a set of dropdown menus (such as "inherit" or "random") will override the values in the dropdown menus. To preview a property on the far right just select it.

Font-Family

☒ sans-serif ☐ inherit ☐ random

Font-Size

☒ Minimum: 12px Maximum: 20px ☐ Minimum: 80% Maximum: 80% ☐ inherit ☐ random

Font Color Families

☒ choose from black and white only ☐ choose from the main colors chosen above (blue) ☐ inherit ☐ random

Use Your Own

Choosing this will override the selection above. One will be randomly chosen for each text color setting if more than one is entered. **NOTE:** You must use two or more *consecutive* text boxes.

Font-Variant

☒ normal ☐ inherit ☐ random

Text-Transform

☒ none ☐ inherit ☐ random

Letter-Spacing

☒ Minimum: 0px Maximum: 0px ☐ inherit ☐ random

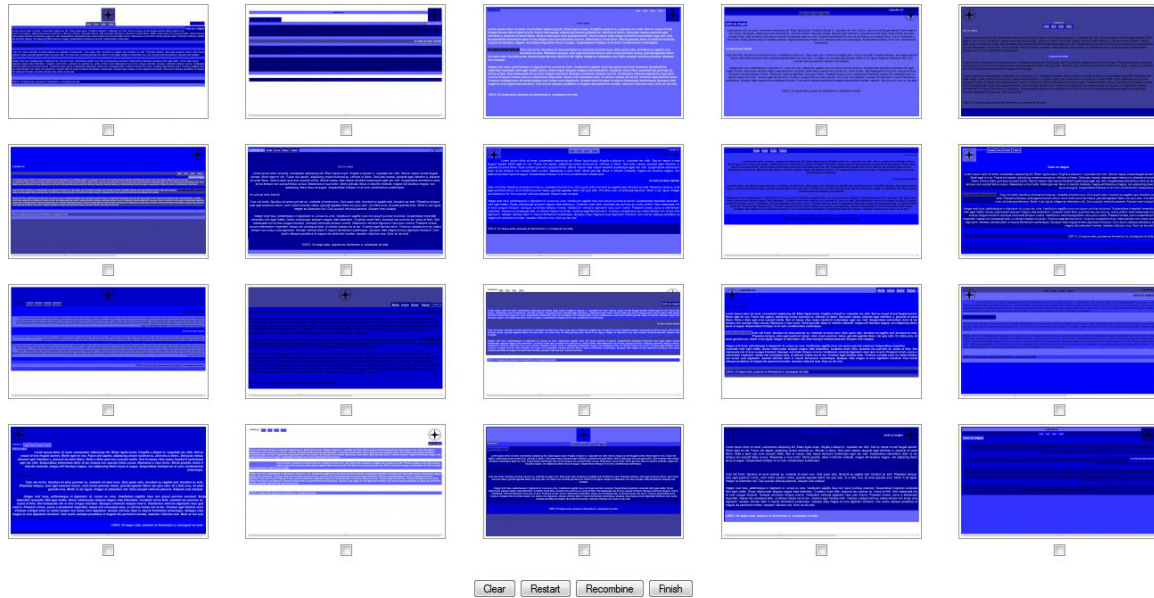
Reset Submit

(b)

Fig. 2.8. More screenshots of the pre-initialization configuration page. (a) Main color scheme and (b) text configuration options.

The evaluation page mainly consists of a 4x5 or 3x5 table (the size depends on the user's screen resolution). Within each table cell is a thumbnail image of a candidate

design with a checkbox below it (see fig. 2.9a). The thumbnails can be clicked to bring up a larger image using a lightbox, which is simply a special way of loading and displaying large images (see fig. 2.9b). Above the table is the number of the current generation; below the table is a row of buttons: clear, restart, recombine, and finish. The table and buttons are contained within a form that is processed by the recombination script once a button is clicked and the form has been validated. "Clear" removes the checkmarks from all previously checked boxes; "restart", "recombine", and "finish" have all been previously described.



(a)



(b)

Fig. 2.9. Screenshots of the evaluation page. (a) Thumbnail images of all candidates. (b) Lightbox image of one candidate.

The finish screen is the most simple. The thumbnail of the design is placed side-by-side with the CSS that created it (see fig. 2.10). The web page that the CSS styled is also made available for download. This is the final screen of the web app.

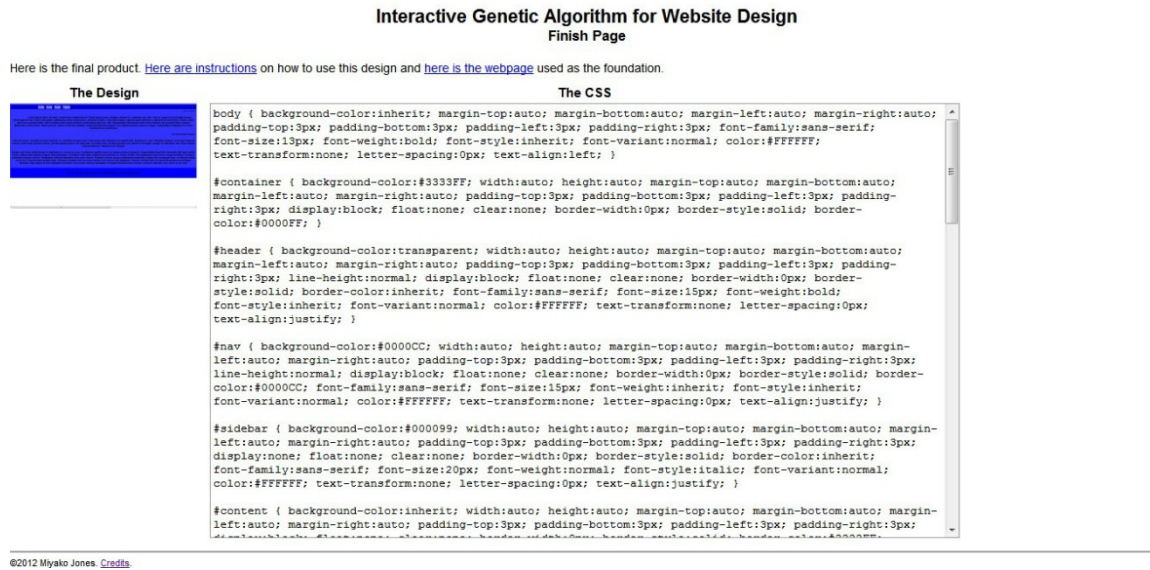


Fig. 2.10. The finish screen.

CHAPTER THREE: RESULTS

There can be a large variation in the results of a typical run due to the large number of options the user can select from. The option that makes the most difference is the page layout style. There are five layouts to choose from or the user can allow the script to randomly choose for him or her. Layouts #4 and #5 have several variations due to the possible ordering of the columns. Also, the total number of candidates in the candidate pool can vary depending on the size of the user's screen resolution. If the height of his or her resolution is less than 768 pixels then the pool will contain 15 candidates; otherwise, it will contain 20.

The page layouts are examples of styles that actual websites use (see fig. 3.1). There is a total of five page sections that can be styled, but not every layout makes use of them all. On average, only four are used: the header, the footer, the navigation bar (nav bar), and the main content area. The fifth section, the sidebar, is only used by a few types of websites (such as online stores) as a secondary navigation bar or to provide additional information to the user.

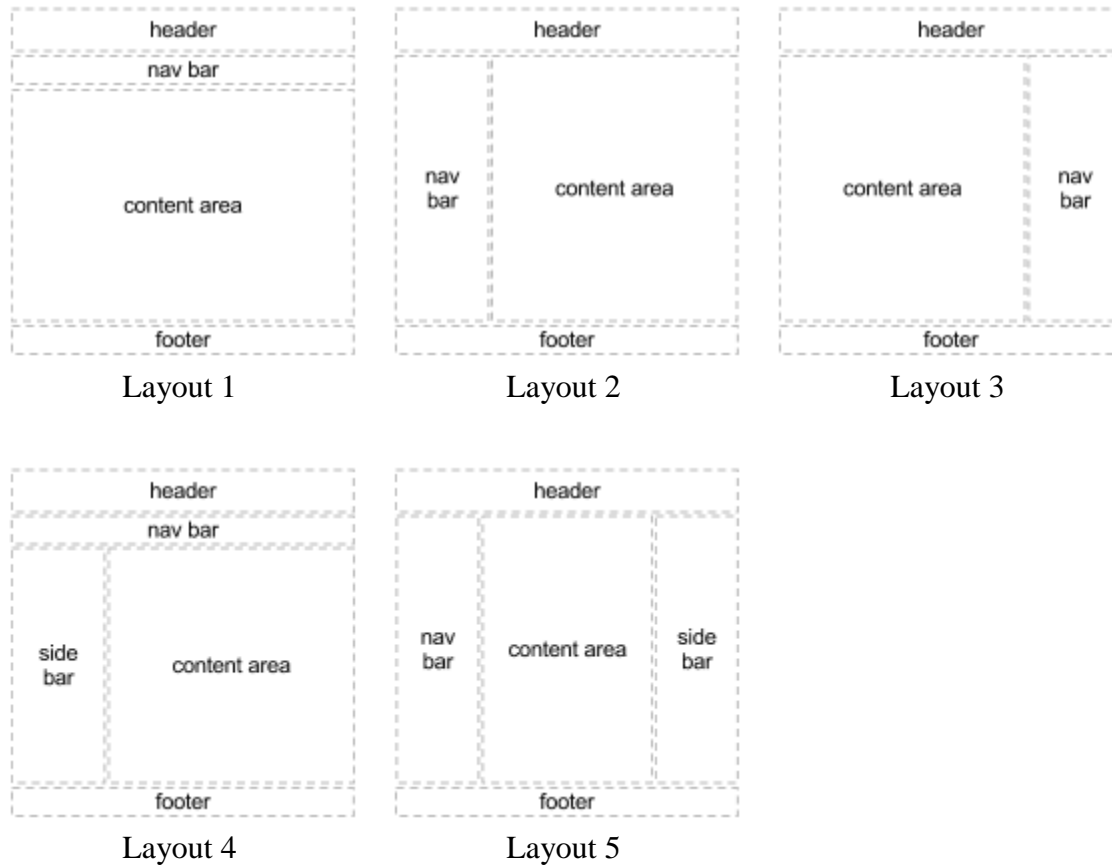


Fig. 3.1. The five IGA layout styles.

Two experiments were conducted while using the IGA for several candidate generations. Both evaluate the IGA's ability to improve the candidate population over time. The results cannot be considered a representation of the candidate quality for a typical web page design session since I was the only participant, but they can be used as a general measurement of the IGA's performance.

3.1. EXPERIMENT 1: ACCEPTABLE CANDIDATES

The goal of experiment 1 was to determine two things: (1) Do large screenshots of a candidate design make a difference in how the user evaluates it? (2) Does the size of the candidate pool make a significant difference in the number of candidates that the user finds acceptable? All layout styles were evaluated for ten generations for both population sizes.

When evaluating candidates using only the small thumbnail images of each candidate, the number of acceptable candidates per generation rapidly reached the number of candidates in the candidate pool for both pool sizes, especially for layout #5 (see fig. 3.2). The number of acceptable candidates for layout #2 fluctuated for both pool sizes as previously selected candidates were discarded in favor of new candidates. Overall, there were far more fluctuations when the size of the candidate pool was 15 than when it was size 20, possibly because better candidates were being generated more often.

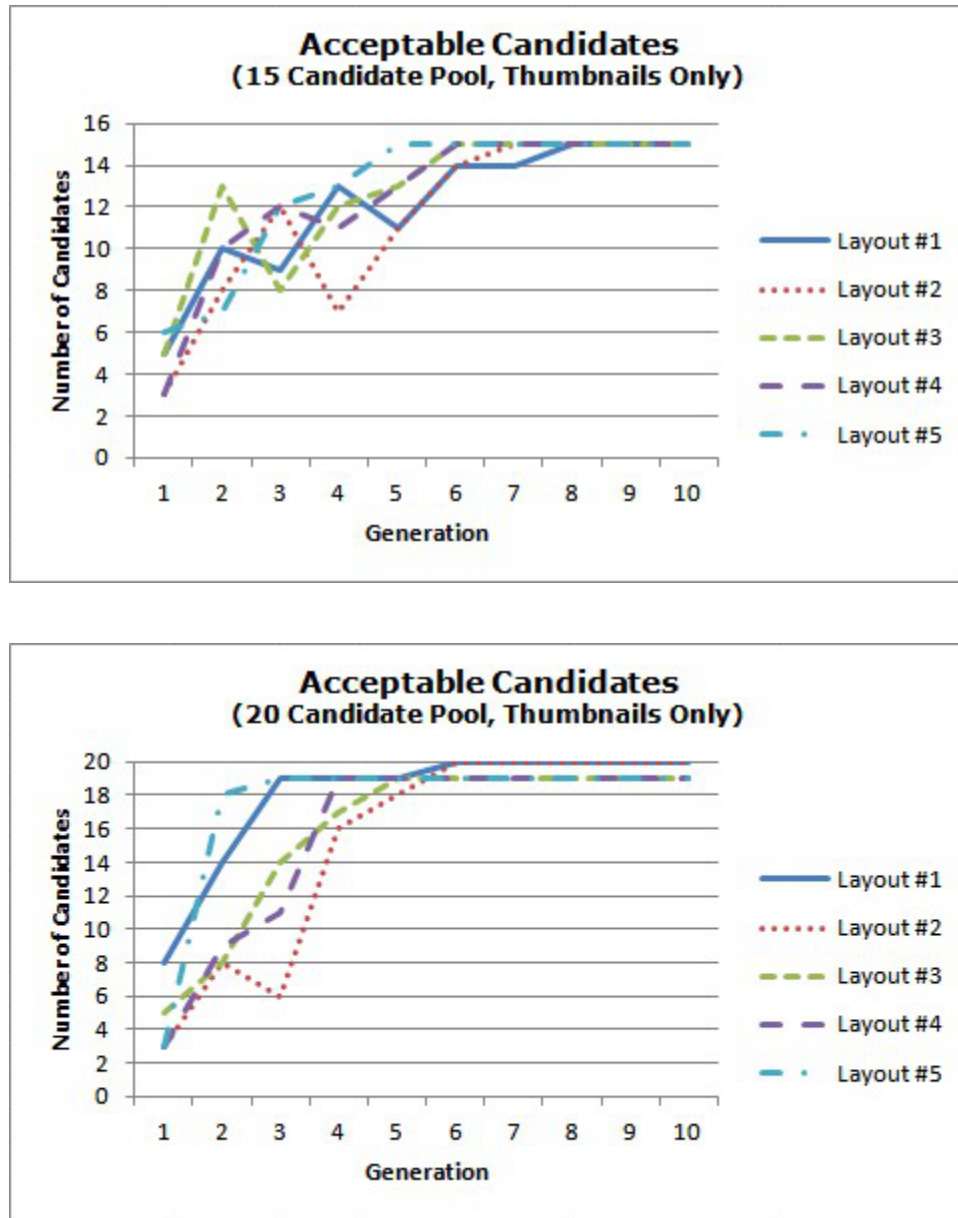


Fig. 3.2. Number of acceptable candidates for pool sizes 15 and 20 using thumbnails only during evaluation.

When each generation was evaluated using both the thumbnail images as well as the much larger lightbox images, the number of acceptable candidates took longer to equal the size of the candidate pool for both generations (see fig. 3.3). There was little

variation from generation to generation in the number of acceptable candidates when layout #1 for both pool sizes. The point where all candidates were acceptable was never reached within ten generations for some layouts, most notably #5 for size 15 and #2 for size 20. While there were some overall fluctuations in the number of acceptable candidates for both candidate pool sizes, there were more fluctuations when the size 20.

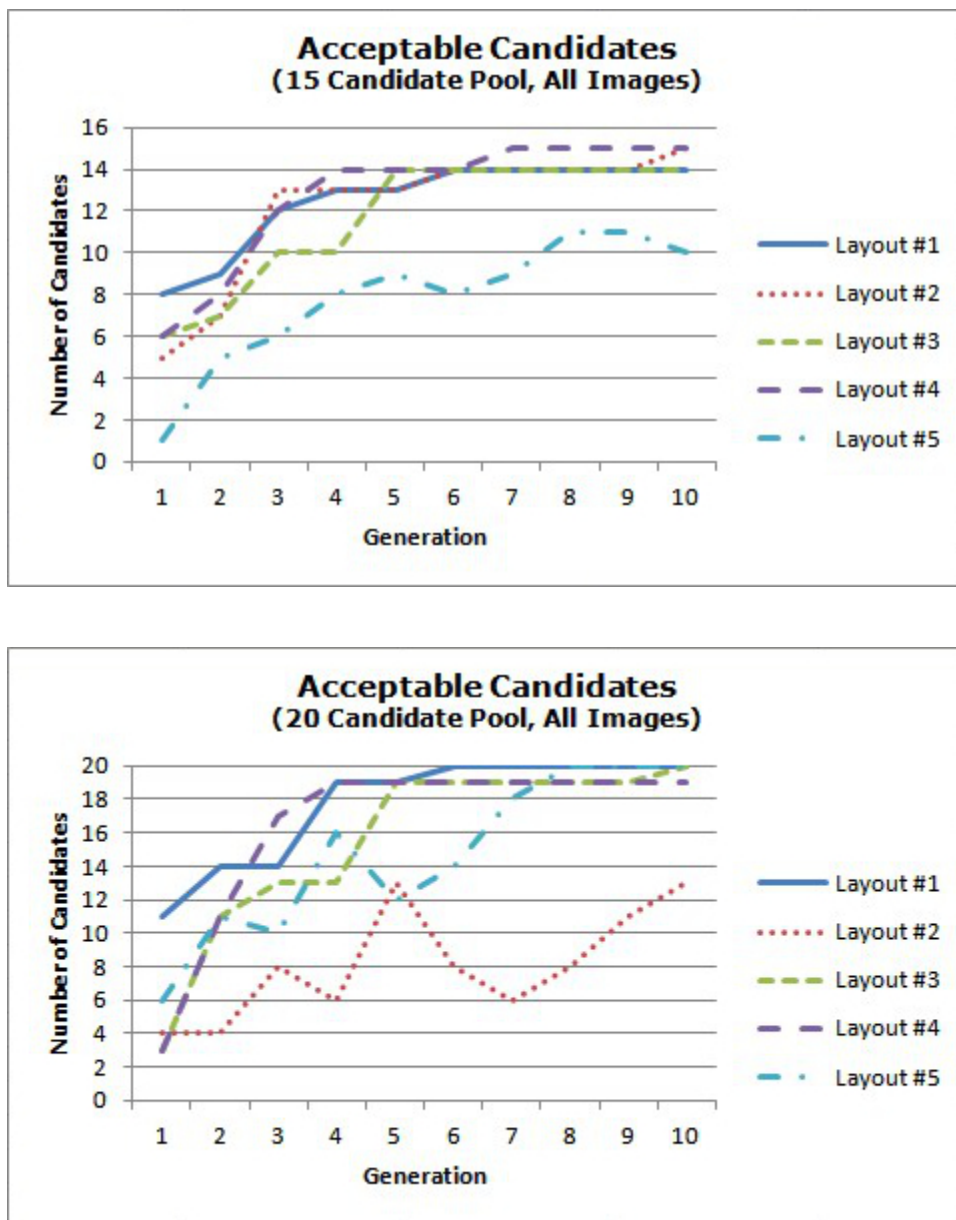


Fig. 3.3. Number of acceptable candidates for pool sizes 15 and 20 using both thumbnails and lightbox images during evaluation.

Overall, the large lightbox images made a significant difference in the number of candidates that were deemed acceptable. Far more detail can be seen when using them and this invalidated candidates that appeared to be fine when viewed as only a thumbnail

image. The number of candidates in the population had little effect when using only thumbnails for evaluation, but it had a significant effect for certain layouts when using both thumbnails and large images. Layout #3 seemed to be the only one barely affected by the size of the candidate population.

3.2. EXPERIMENT 2: AVERAGE FITNESS LEVEL

Experiment 2 was meant to evaluate the overall effectiveness of the recombination step. Is the quality of the candidates improving over time? If so, by how much? The average fitness level per generation was calculated from the data recorded when evaluation was performed using both thumbnail images and the lightbox images for both candidate population sizes. There is data only through generation nine for each layout style and population size because recombination was not performed on generation ten, though the number of acceptable candidates was recorded.

For layouts #1, #3, and #4 there was a steady upward trend over the course of the session for both population sizes (see fig. 3.4). Layout #5's trend was also similar for both, but the rate at which the average fitness increased fluctuated. Layout #2 produced significantly different results for a population of size 15 than it did for one of size 20. The average fitness level was much higher with very little fluctuation for population size 15.

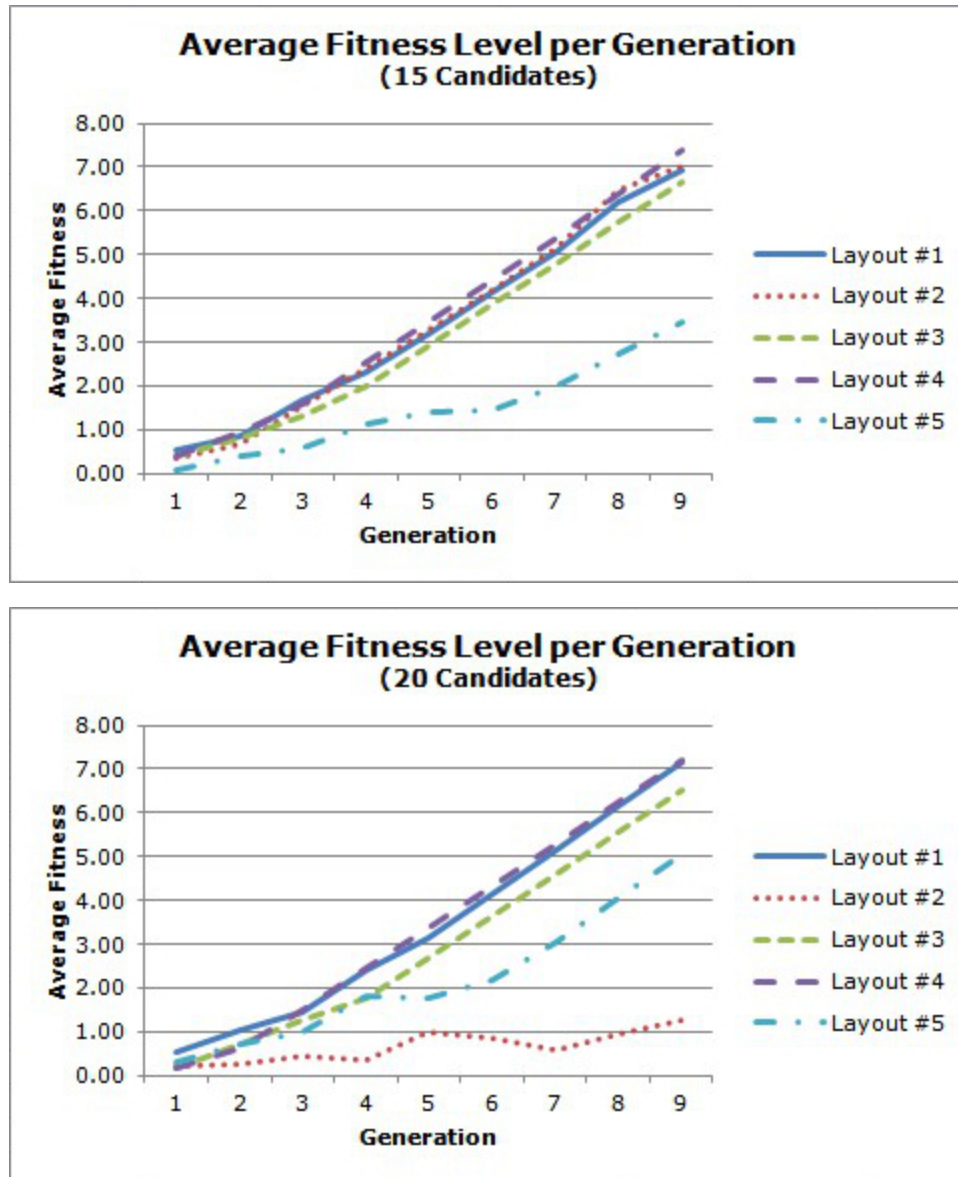


Fig. 3.4. Average fitness for candidate pool sizes 15 and 20.

Once again, there was evidence that the size of the population greatly affects the quality of the candidates for layout #2 while layout #3 was barely affected at all. The results for layout #5 seemingly contradict the number of acceptable candidates per generation as there were far more differences overall in the number of acceptable candidates than in the average fitness levels between the different population sizes.

When looking solely at the results for population size 20, the differences are very noticeable. For example, between generations 4 and 5 there was a four candidate drop, but the average fitness level dropped by only 0.05. One possible explanation for this discrepancy is that the candidates discarded during the evaluation of generation 5 were not in the pool long enough to significantly affect the average fitness of generation 6. In other words, majority of the discards were brand new candidates.

3.3. FINAL THOUGHTS

While layout complexity does, indeed, affect the results, some of them were surprising. The most complex layout, #5 (which also has the largest number of variations), consistently produced, on average, less satisfactory candidates, but more than half of the candidates were acceptable for both population sizes by generation 10. I had expected the number of acceptable candidates to be slightly lower. However, the simplest layout, #1, produced a large number of satisfactory candidates very quickly for both population sizes, which was what I had expected.

The fitness of the candidates produced using layout #5 with a population size of 15 was in line with what I had expected as it is typically much lower than the others, but this is not the case when the population size is 20. Layout #1 produced satisfactory candidates at a high rate for both sizes, which was, again, expected.

The ranges of valid values for the layout-related CSS properties that cannot be changed by the user also seemed to have an affect on the quality of the candidates. The properties for layout #1's selectors have the fewest possibilities while the properties for layout #5's have the most. Therefore, it is more likely that layout #5 will produce an

unacceptable candidate compared to any of the other layouts due to the larger number of possibilities. By reducing the range of valid values for some of the layout's properties, it is possible to improve the overall quality of candidates produced, but it will be at the expense of variety.

CONCLUSIONS AND FUTURE CONSIDERATIONS

There are several things that can be done to improve user satisfaction with the IGA. It is possible to improve the quality of the candidates, to improve the candidate evaluation experience, or to reduce the number of generations it takes for the algorithm to produce a candidate that the user decides to choose. One way to improve user satisfaction without altering any of the algorithm's stages is to store session information on the user's computer so that they can resume their session at a later time. This alteration will make the most difference if no other changes are made as it currently takes many generations for an completely acceptable candidate to be produced (if at all).

I believe that any of the following changes would have a significant impact on the overall user experience:

1. Generate candidate designs based upon the type of website the user wants to create.
2. Use color theory to suggest harmonious color schemes.
3. Allow identical tags in different page sections to have different values.
4. Alter the crossover and/or mutation rate over time.
5. Modify the restrictions for crossover point selection.

Candidate quality improvements will come primarily from alterations to the initialization stage. If the user is able to choose a website "style" versus a page layout, then the candidates generated by the IGA are more likely to be what they want. It is possible that the user has an idea in mind that is based on a website he or she has previously visited. If the algorithm generates candidates based upon a website style or type (such as a blog or a online store), they are much more likely to be what the user expected. Also, providing guidance on choosing colors that is based on color theory

would greatly help the algorithm generate quality candidates as the colors being used can have a significant impact on how the user evaluates a design.

The evaluation stage can be improved by increasing the quality and/or variety of the candidates. If the quality of the initial population is improved, then it will take less time for the user to decide on a single candidate. If the variety of these same candidates is also improved, the user will find the web app more useful. Candidate variety can be improved by allowing the same selector in different sections of the web page to take on different CSS property values (a paragraph HTML tag in the content area can have different values than one in the footer), by removing one or both of the crossover restrictions (the "selector boundary" restriction or the "entire property swap" restriction), or by altering the crossover and/or mutation rates (e.g. increasing one or both rates). More than one of these changes can be applied at the same time.

There are several other additions and changes that could be made to the IGA in order to enhance the user overall experience. Occasionally, when a crossover occurs between two candidates, there is no visible difference in the appearance of the children. They, essentially, appear to be clones. This occurs when the swapped CSS property values are overridden by later values that are identical to the parent's values. When this occurs, the variety of the candidate population seemingly decreases from one generation to the next. There is no simple way to correct this behavior since it is controlled by the CSS cascade.

Another issue is that it is possible that two identical candidates will be swapped, which wastes time and processing power. There is a way to prevent this behavior by testing for equivalency before performing crossover and choosing an alternate set of

potential parents. However, if the candidate population has nearly converged to a single candidate, then this process will likely be more wasteful than swapping two identical candidates. If it *has* converged, then this process will never complete as all of the candidates in the population are identical.

It is currently impossible for more than one user to use the IGA at a time. The stylesheets and candidate screenshots (both full-size and thumbnail) use a specific naming style regardless of the source. Stylesheets are named *style* and are numbered from 0 to one less than the maximum population size while full-size screenshots and thumbnails are both named *candidate* (they are saved to separate folders) and are numbered from 1 to the maximum population size. One way to correct this problem is to prepend the user's PHP session ID to the beginning of each filename since this ID is unique for every user.

Though there is room for improvement, the IGA successfully generates a variety of candidates for the user to choose from. The user's preferences are taken into account from start to finish and he or she is directly involved in the evolution of the candidate population. Although the "perfect" web page design may never be generated, the user can explore many others that might help him or her develop their ideal design.

REFERENCES

- Andrew, Paul. "15 Responsive CSS Frameworks Worth Considering." *Speckyboy Design Magazine*. 17 Nov. 2011. Web. 13 Oct. 2012.
<<http://www.speckyboy.com/2011/11/17/15-responsive-css-frameworks-worth-considering/>>.
- Bauerly, Michael, and Yili Liu. "Evaluation and Improvement of Interface Aesthetics with an Interactive Genetic Algorithm." *International Journal of Human-Computer Interaction* 25.2 (2009): 155-66. Web. 02 Feb. 2010.
<<http://dx.doi.org/10.1080/10447310802629801>>.
- Calborean, Horia, and Lucian Vintan. "An Automatic Design Space Exploration Framework for Multicore Architecture Optimizations". *Roedunet International Conference (RoEduNet), 2010 9th*. 24-26 June 2010, Ed. Brad Remus. Sibiu, Romania. *IEEE Xplore*. Web. 28 Nov. 2012.
<http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5541567>.
- Choueiry, Berthe Y. "Blind Search Algorithms." *Homepage*. University of Nebraska-Lincoln. Web. 20 Sep. 2012. <<http://www.cse.unl.edu/~choueiry/S03-476-876/searchapplet>>.
- Chaney, Allison. "Breadth-first search." *Homepage*. Princeton University. Web. 20 Sep. 2012. <http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Breadth-first_search.html>.
- Chaney, Allison. "Depth-first search." *Homepage*. Princeton University. Web. 20 Sep. 2012. <http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Depth-first_search.html>.

- Cho, Sung-Bae. "Towards Creative Evolutionary Systems with Interactive Genetic Algorithm." *Applied Intelligence* 16.2 (2002): 129-38. Web. 14 Oct. 2010. <<http://dx.doi.org/10.1023/A:1013614519179>>.
- Jones, M. Tim. "Introduction to Genetic Algorithms." *AI Application Programming*. Hingham, MA, USA: Charles River Media, 2003. 115-149. *ebrary*. Web. 8 Sep. 2010. <<http://site.ebrary.com/lib/umich/Doc?id=10061221>>.
- Karásek, Jan, Radim Burget, and Ondřej Morský. "Towards an Automatic Design of Non-Cryptographic Hash Function". *International Conference on Telecommunications and Signal Processing (TSP), 2011 34th. IEEE Xplore*. Web. 27 Nov. 2012. <<http://dx.doi.org/10.1109/TSP.2011.6043785>>.
- Koza, John R. *Genetic Programming, Inc. Home Page*. Jul. 8 2007. Web. 28 Jun. 2010. <<http://www.genetic-programming.com>>.
- Luger, George F. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. 5th ed. Addison-Wesley, 2005. Print.
- Mackinlay, Jock D. "Search Architectures for the Automatic Design of Graphical Presentations." *Intelligent User Interfaces*. Eds. Joseph W. Sullivan and Sherman W. Tyler. New York: ACM, 1991. 281-292. PARC Homepage. Web. 27 Nov. 2012. <<http://www2.parc.com/istl/groups/uir/publications/items/UIR-1991-07-Mackinlay-IUI-Search.pdf>>.
- Marshall, David. "Heuristic Search." *Homepage*. Cardiff University. Web. 22 Sep. 2012. <<http://www.cs.cf.ac.uk/Dave/AI2/node23.html>>.

- Monmarche, N., et al. "Imagine: A Tool for Generating HTML Style Sheets with an Interactive Genetic Algorithm Based on Genes Frequencies". *1999 IEEE International Conference on Systems, Man, and Cybernetics (SMC'99)*. Tokyo, Japan. Web. 24 Jan. 2010.
<<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.6918>>.
- Oliver, A., N. Monmarch, and G. Venturini. "Interactive Design of Web Sites with a Genetic Algorithm". *IADIS International Conference WWW/Internet*. Web. 23 Oct. 2011. <<http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.1950>>.
- Porebski, Bartosz, Karol Przystalski, and Leszek Nowak. "Introducing Symfony, CakePHP, and Zend Framework." *Building PHP Applications with Symfony™, CakePHP, and Zend® Framework*. Indianapolis: Wiley, 2011. *ProQuest Tech Books*. Web. 02 Oct. 2012.
<<http://proquest.safaribooksonline.com/book/programming/php/9780470887349>>.
- Pressman, Roger S., and David Lowe. "Functional Architecture." *Web Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 2009. 279-286. Print.
- Pressman, Roger S., and David Lowe. "What is Web Engineering?" *Web Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 2009. 12-16. Print.
- Raggett, Dave, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification*. W3C, 1999. Web. 24 Dec. 2010. <<http://www.w3.org/TR/1999/REC-html401-19991224>>.
- Reeves, Colin R., and Jonathan E. Rowe. "Basic Principles." *Genetic Algorithms - Principles and Perspectives: A Guide to GA Theory*. Eds. Ramesh Sharda and

- Stefan Voss. Secaucus, NJ, USA: Kluwer Academic Publishers, 2002. 19-61.
Operations Research/Computer Science Interfaces. *ebrary*. Web. 8 Sep. 2010.
<<http://site.ebrary.com/lib/umich/Doc?id=10067477>>.
- Reeves, Colin R., and Jonathan E. Rowe. "Introduction." *Genetic Algorithms - Principles and Perspectives: A Guide to GA Theory*. Eds. Ramesh Sharda and Stefan Voss. Secaucus, NJ, USA: Kluwer Academic Publishers, 2002. 1-18.
Operations Research/Computer Science Interfaces. *ebrary*. Web. 8 Sep. 2010.
<<http://site.ebrary.com/lib/umich/Doc?id=10067477>>.
- Srinivas, M., and Lalit M. Patnaik. "Genetic Algorithms: A Survey." *Computer* 27.6 (1994): 17-26. *IEEE Xplore*. Web. 01 Sep. 2010.
<<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=294849&isnumber=7284>>.
- Tyugu, Enn. "Search." *Algorithms and Architectures of Artificial Intelligence*. Eds. J. Breuker, et al. Vol. 159. Amsterdam: IOS Press, 2007. *ebrary*. Web. Frontiers in Artificial Intelligence and Applications. 247 vols. 20 Sep. 2012.
<<http://site.ebrary.com/lib/umich/docDetail.action?docID=10196612>>.