Genetic Algorithms for a Multiagent Approach to the Game of Go

碁

by

Todd Blackman

B.S. Computer Engineering, University of Kansas, 2000

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

____
Dr. Arvin Agah, Professor in Charge

____
Dr. Victor Frost, Committee Member

____
Dr. Costas Tsatsoulis, Committee Member

Date Defended: April 25, 2003

Abstract

Many researchers have written or attempted to write programs that play the ancient Chinese board game called *go*. Though some of these programs play quite well compared to beginners, few play extremely well, and none of the best programs rely on soft computing AI techniques such as genetic algorithms or neural networks. This thesis explores the advantages and possibilities of using genetic algorithms to evolve a multiagent go player. We show how each individual agent plays poorly, while the agents working together play the game significantly better.

Acknowledgments

I would like to thank Dr. Arvin Agah for all of his advice and cajoling. His intelligence and humor made my research experience both enjoyable and rewarding. I would also like to thank Dr. Costas Tsatsoulis and Dr. Victor Frost, both committee members. Dr. Tsatsoulis has been a great mentor and teacher during my years at the University of Kansas while Dr. Frost has been an inspiration and has impacted my future in countless ways. I would also like to recognize the significant use of code from *Genetic Algorithms in Search, Optimization, and Machine Learning* [6] and *Numerical Recipes in C: The Art of Scientific Computing* [15]. On a more jovial note, I would like to thank Linus Torvalds for allowing me the opportunity of *not* using any Microsoft (TM) products including, but not limited to, Word (TM), Excel (TM), Powerpoint (TM), Internet Explorer (TM), and Windows (TM). Revision control was handled by CVS, and this document was typeset using LaTeX. Hayley Chapnick and Mara Reichman both gave of their time to proofread this work. Finally, I would like to thank Kristy Blackman for dealing with all of the time I spent in my office working on my computer and Shawn Steiman for inspiring me to finish my thesis in a timely manner.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Games have often been used to test new concepts in artificial intelligence because of their relative simplicity compared to other more complex possibilities such as simulations and real-world testing. Go has the potential to excel as a testbed for AI concepts because of the complexity of the tactics and strategies used to play the game well. These complexities resemble real-world problems better than most other games. Brute-force search cannot be used exclusively to play this game, as in other games, because of go's huge branching factor which starts out at 361 at the beginning of a game and approximately decreases by one after each move.

With pure search ruled out as a viable method for playing go, one must turn to more intelligent methods such as pattern recognition or rule-based deduction. Complexity often plagues go programmers because of the intricacies of how a player must think about the game—often remote locations on the board influence a local situation. Sometimes, what one would hastily consider the best move really reveals itself as the worst move because of global concerns on the board. Current go programs play at only the level of a skilled novice, and we believe that these limitations exist because of the programs' architectures and their insistence on using only traditional methods such as pattern-matching, hard-coded rules in computer code, and minimax with alpha-beta pruning.

What we propose is that programs should play go using relatively simple agents that combine to play the game well. Traditional methods have their place in go programs, but *to play an abstract and multi-faceted game one must use an abstract and multi-faceted approach.* Genetic algorithms have been employed to play complex games, but these genetic algorithms often use evolved values that are too low-level to allow the program to attain the skill required to play well. By *low-level*, we mean that these values allow for the evolution of useful information such as patterns or algorithmic code, but to play the game on a professional level one would need too many of these individual pieces of information. Analogously, it would be like creating a neural network with 3×19^2 inputs (representing the 19^2 board locations and the three possible states for each location: white, black, empty) and 19^2 outputs. Training an artificial neural network of this size will remain inconceivable for quite some time. Likewise, trying to evolve a set of rules using a genetic algorithm would fail in much the same way. Too many rules exist, and evolving them would take too long.

Our program and approach differ from most current go programs. Other programs are extremely complex, representing huge amounts of go knowledge. They eventually become unwieldy, difficult to maintain, hard to follow, and tricky to improve upon.

The motivation for this work is thus to study whether a set of relatively simple agents can each look at the problem from their own perspective after which genetic-algorithm-evolved weights will allow the agents' solutions to be summed together to produce a final solution. This approach exchanges the ability to fine-tune the program with the ability to incorporate more agents, and thus more knowledge, in a consistent and scalable way. Our research will illustrate a novel multiagent approach to playing games that uses a multilayer network to suggest moves based on the moves suggested by each individual agent.

The problem is therefore to develop a set of agents that generate a value for each location on the board (higher values representing a more highly recommended move). These values are entered into a matrix such that each location on the go board corresponds to a place in the matrix. These matrices are then normalized and combined non-linearly using genetic-algorithm-evolved weights. The summation network architecture, shown in Figure 1.1, resembles a neural network in that the resulting matrix of values is generated from a weighted sum of a set of weighted sums. The resulting move to play will then be either the highest value in the matrix or chosen probabilistically with higher values receiving a greater

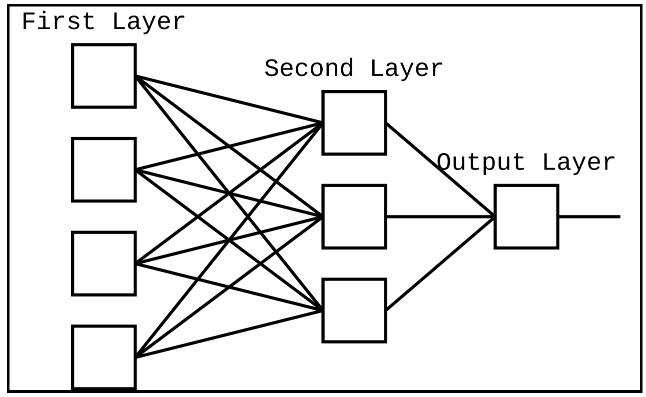


Figure 1.1: Summation Network Architecture.

probability of being chosen. Thus, our goal is to:

1. Show that as the genetic algorithm evolves, the program plays better.

2. Show that as more agents are added and the chromosomes evolved, the program plays better.

3. Show that simple agents can be used to achieve better performing *meta behavior*.

4. Illustrate a novel multiagent approach to programming games.

These considerations are obviously limited by our ability to program knowledge about go; consequently, we have an additional goal of becoming better players so as to program the game better. Also, we tried to develop agents that were not exceedingly complex. They should have specific and well-defined expertise and a clear focus.

1.1 Organization

Chapter 2 begins with information about the game of go including the rules of the game, a few direct implications of the rules, basic concepts, how to score a game, and how players are ranked. This is followed by a description of some techniques relevant to this thesis. Following these, we consider how AI relates to games, and in particular, to the game of go.

The next chapter (Chapter 3) explores the design of the program written for this thesis. It explains all of the main components of the program and provides a top-level view of the program's architecture.

In Chapter 4 we explain all of the experiments performed along with the results of these experiments. In addition to the results, some conclusions are drawn as to the relevance of the data and what the interpretation of the data is. The next chapter wraps up this thesis and summarizes the contributions, limitations, and future of this work.

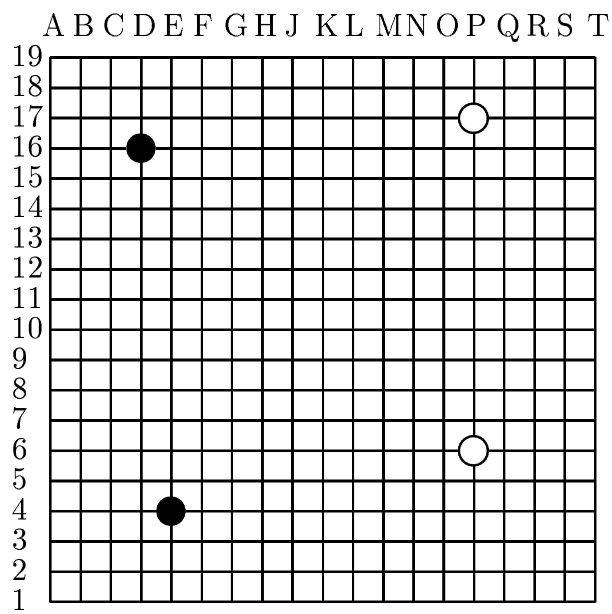
Chapter 2

Background and Related Work

2.1 The Game of Go

The game of go (also called goe, igo, baduk, wei-chi, weichi, weiqi, wei-qi, etc.) is a board game of perfect information[1] played by two opponents on a 19×19 grid as shown in Figure 2.1. Each player takes turns placing his or her piece (called a *stone*) on an intersection starting with the individual playing black. The opponent, as white, places his or her stone, and the game continues until both players pass their turns in succession. No stone can be moved unless it is captured (as will be explained later), and all stones are completely equal in power unlike pieces in games such as chess.

[1] *Perfect information* is a term used to describe games that allow the players to see the entire state of the game at all times. No guessing or probability is involved as in games such as backgammon or bridge which have uncertainty and hidden state respectively.



Notice that the letter I is not used which is to prevent confusion with the letter J when transcribing games.

Figure 2.1: A Go Board.

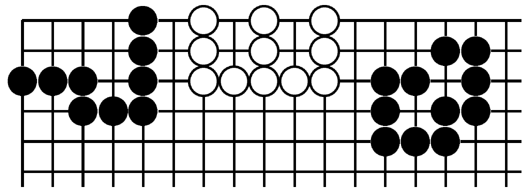


Figure 2.2: Surrounded Territory.

2.1.1 Surrounding Territory

The goal of the game is to surround more territory than your opponent with a secondary goal of capturing your opponent's stones. Each surrounded intersection or captured stone is worth one point[2]. In general, surrounding territory is considered much more important by anyone versed in go. Figure 2.2 shows three examples of surrounded territory: The black group on the left surrounds nine points, the white group surrounds four points, and the right-most black group surrounds two points.

2.1.2 Capturing

As stated, there is also a secondary way to gain points—capturing the opponent's stones. To capture a single stone, one must play on all adjacent intersection points that are at right angles to the stone(s) to be captured. Figure 2.3 shows three examples of a white stone about to be captured by a black stone if black were to play on the locations marked A. The white stone is *in atari*.

[2]An exception exists when a stalemate condition arises as will be explained later

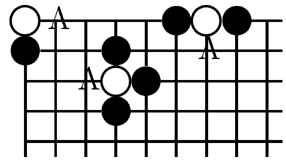


Figure 2.3: White Stones to be Captured on Next Move (in Atari).

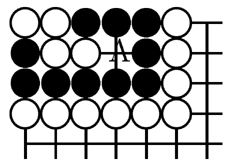


Figure 2.4: Capturing.

To capture groups[3] one must play on all the liberties[4] of the group. Figure 2.2 illustrates three disjoint groups, and Figure 2.4 shows an example of a possible state of the board during a game. In this figure, if it were white's turn, she could play at the location marked *A* to capture ten black stones. On the other hand, if it were black's turn to play, he could also play at *A* to capture four white stones. Furthermore, if white were to play at location B in Figure 2.5, then white would only capture *seven* stones as the two black stones at the top of the board are *not* part of the black group below it.

[3]A group is defined as a set of stones that connect adjacently to each other through the straight lines on the board (i.e., at right angles). *Diagonals* do not count.

[4]A liberty is simply an empty location adjacent to a group.

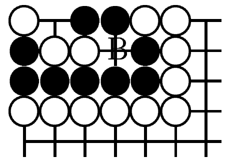


Figure 2.5: Capturing II.

2.1.3 Eyes

It follows that any group is *unconditionally* safe if it can partition itself into at least two sections (also called *eyes*). Figure 2.2 illustrates multiple groups that have two eyes each—the two rightmost groups. These groups are unconditionally safe because if the opponent plays in an eye with a single intersection inside that eye, then it commits suicide (all of its liberties are taken). One can logically rationalize the unconditional safeness of a group with two or more eyes by imagining that to kill the group, one would have to play in all eyes at the same time (after surrounding the group first) which is of course illegal. If the eyes are too big, one's opponent could create a living group inside an eye and then the eye could become useless. As an aside, the group on the left in Figure 2.2 has an uncertain living ability; it is neither alive nor dead as it stands.

2.1.4 Live and Dead Stones

Surrounding territory is crucial while playing go, but there is a very important twist that can make what seems like one's territory actually one's opponent's.

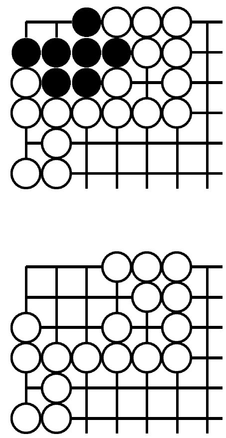
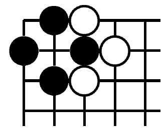


Figure 2.6: Dead Group.

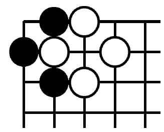
If a group of stones (at the end of the game after both sides pass in succession) could not possibly survive an attack (i.e., it does not have two eyes or the ability to make them if pressed), then that group is removed from the board and given to the opponent. After playing many games, one soon learns how to identify hopelessly dead groups of stones—if a disagreement arises about whether a group is alive, then the game continues. Figure 2.6 shows a black group that is hopelessly dead and the resulting board fragment after it is removed.

2.1.5 Rule of Ko

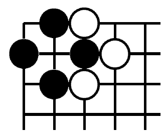
The last important concept in go is that of the rule of ko. This rule states that no board state may be repeated. Stated another way, livelock is not allowed to occur. The sequence of plays in Figure 2.7 illustrates an example of what could happen if this rule were *not* in effect. The first move by white ($S_0 \rightarrow S_1$) captures



S_0



S_1



S_2

Figure 2.7: Example of Go Without the Ko Rule.

a black stone, while the second move ($S_1 \to S_2$) captures a white stone. This second move ($S_1 \to S_2$) is illegal, and black *must* play somewhere else. One can see that without the ko rule, a livelock situation would arise and both sides would continually gain the same number of prisoners (stones).

2.1.6 Seki (Stalemate)

Seki can be viewed as a localized stalemate condition. In the game of go, there are situations when neither side can count his or her territory because both sides would have dead groups *if* they played first. Figure 2.8 illustrates this condition. If it were white's turn, she could not play at A (suicide), while playing at C would fill in her own eye. The only option is for white to play at B, but that would allow black to play at C on the next turn, capturing the white group. Likewise, if it

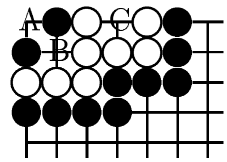


Figure 2.8: Seki.

were black's turn to play, C would be suicide and A would be filling in his own eye. A play by black at B would allow his small group to be killed with a white play at A. Thus, the three locations A, B, and C are not counted as territory.

2.1.7 Scoring

Many variations exist for scoring finished go games, but for simplicity a player's score is calculated by first removing dead groups (which become prisoners), then counting the number of captured prisoners, and finally counting the number of intersections completely surrounded by one's own color. In many games the person to play second may get additional points called a *komi* to compensate for going second which can vary from 0.5 to 5.5 points.

2.1.8 Other Board Sizes

Go can be played on boards of any size, but historically, games have been played on boards of size 19, 17, 13, and nine (for a total of 361, 289, 169, and 81 intersections respectively). Nine is used to teach beginners and is also often used in computer go games because it has a smaller search space by orders of magnitude. Boards of size 17 are usually used when one wants to have the essence of a full game but does not have the time for a full 19×19 board. A 9×9 board, for example, has a very different character than a 17×17 or a 19×19 board.

2.1.9 Go Player Ranking

Go has a well standardized hierarchy of skill levels that allow players to compete on a fairly equal basis with standardized handicaps. A complete beginner that has played a game and knows the rules starts off at *30 kyu*. The scale progresses to *one kyu* which is the best kyu ranking one can attain. After this, the scale continues at *one dan* up to *nine dan*, the highest amateur ranking possible. Thus, for the kyu ranks, lower values imply a better player, while for the dan ranks, higher values imply a better player. To confuse the issue, professionals rank themselves on the *dan* scale as well from one to nine, but their rankings are usually considered stronger. Thus, a *five dan* professional and a *five dan* amateur would not usually be able to play an equal game (with no handicap). Further complicating the matter, different countries and groups may not completely coincide with each other on strength. For example, a Korean *eight kyu* might not be equal to a British *eight kyu*. Computer programs are often given honorary diplomas of a certain level, but these can be misleading as the programs often play well but make horrible mistakes every once in a while.

2.2 Relevant AI and Computational Techniques

Numerous AI and computational techniques exist that find uses in go programs. Search techniques, neural networks, thread pools, multiagent systems, and genetic algorithms make up what can be considered relevant paradigms to programming the game of go [12, 19, 20].

2.2.1 Search Techniques

Many methods of search exist in the repertoire of most computer scientists. Quite a few are a variation on either breadth-first or depth-first search. These methods include uniform-cost, iterative deepening, bidirectional, and depth-limited searches. Canonical breadth-first search expands a new tree layer at every iteration while depth-first search expands one element from the next layer at every iteration. Uniform-cost search, a variant of the breadth-first search, expands the next cheapest node at every iteration, and iterative deepening search is simply breadth-first with the number of layers increased at each iteration. Bidirectional search attempts to search from both the goal and the starting state simultaneously. Finally, depth-limited search is a variant of depth-first search that includes a provision for the maximum depth that will be searched before backing up.

These search methods can be improved upon by incorporating knowledge about the problem space to help make the search more efficient. The simplest informed method is a greedy search that always expands the node that appears closest to the solution. Another method is A^* and its close relative IDA^* (*iterative deepening* A^*) [19]. In A^* the next node to be expanded is the node that has the lowest value for a variable ϕ. This variable ϕ is defined as the cost from the initial node to the current node in question plus the estimated cost of the best path from that node to the goal.

2.2.2 Neural Networks

A neural network can be viewed as a random search method that yields a function that sometimes cannot be found by more traditional methods. Neural networks are often composed of multiple layers of neurons, and each neuron in each layer can be connected to all of the neurons in the immediately adjacent layer. Each connection has a weight associated with it, and each input neuron receives some part of an input signal which is passed through a non-linear activation function (which determines if the neuron will fire). A neuron that fires has its output signal multiplied by the aforementioned weights which is then directed into all the connected neurons in the next layer. Each neuron in this layer receives a signal from each input neuron, and these signals are then summed and once-again passed through an activation function to determine its output. This process continues for all neurons in all layers until an output is received on the output layer of the neural network.

Many paradigms exist for training neural networks, but the most common is the backpropagation technique which compares the output of the network with a *correct* training example. The delta between the expected and actual output is backpropagated through the network to modify the weights between the neurons. Other methods exist to modify the weights [16].

The usefulness of neural networks is heavily influenced by a number of factors: the number of neurons, the number of layers, the choice of whether to allow connections within a layer, the choice of whether to allow connections back to previous layers, the choice of training data, the learning rate, and the choice of the training method. These all can affect the quality of the resulting network. There is much trial-and-error involved in neural network design.

2.2.3 Multiagent Systems

As described in Gerhard Weiss's book, "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in the environment in order to meet its design objectives" [20]. Though there is much disagreement on the exact definition of an agent, most agree that agents are indeed autonomous. Intelligent agents have the characteristics of agents but also have intelligent traits such as reactivity, pro-activeness, and social ability [20]. A robot capable of interacting in its environment might be considered an intelligent agent, while the thermostat in one's home would not be.

There are many issues that must be resolved if one wants to create a multiagent system. One must consider what kind of information the agent will have about the environment that it exists in. A computer program running a single thread for each agent is quite different from a robot traversing ice-sheets at the poles of the earth. Is the environment real or virtual? Will the agent receive all inputs through a socket, via shared memory, or by way of an external bump sensor? All of these questions are important to designing a multiagent system. Closely related to this concept of an agent's environment is the idea of an *ontology* which is, "... a specification of the objects, concepts, and relationships in an area of interest" [20]. All agents must exist within some framework.

Agents that simply exist in an environment that they can sense are different from cases where a method of communication becomes important to consider. If the agents exist in a virtual world on a single computer they might communicate via shared memory or via sockets, while distributed agents might communicate over a wireless network or even with physical means such as actual speech.

Being able to communicate is of little use if one does not have a common language and protocol for exchanging messages, thus these concerns arise and must be considered by a multiagent system designer. Many languages and protocols exist for agent communication [8], but the important idea is that these issues must be carefully considered for the application at hand.

A high-level goal must exist that enables the agents to actually achieve something useful. Even if no single entity has established a clear plan or goal, often a goal is inherent in the behavior of each single agent. For example, one could argue that a goal of human society is to survive. This goal is exemplified in each individual human-being's actions. Analogously, computer agents interact together to produce some type of useful output. Agents can be cooperative or self-interested which both lead to different types of agent interactions. Cooperative agents may actually negotiate an agreed upon goal, while self-interested agents might achieve their own personal goals leading to a net benefit to at least themselves.

2.2.4 Genetic Algorithms

Genetic algorithms are search methods that approximate biological genetics (i.e., simulate evolution) in an attempt to find a solution or goal for a particular problem [11, p. 1–6]. Essentially, a genetic algorithm (or GA) begins with the creation of a set of entirely random chromosomes of alleles. These "arrays of bits" are translated into parameters or data that can then be tested to see how well they approximate the solution that is being searched for. This function that finds how well an individual chromosome performs is called a *fitness function* or *objective function*.

GAs progress by first creating a population of randomly generated chromosomes. The fitness of each of these chromosomes is calculated and then pairs of chromosomes are picked (with higher fitness values more likely to be picked).

These pairs possibly undergo crossover and/or mutation. The resulting chromosome then becomes a new member of the new population. The process repeats itself for each generation until a chromosome with some minimum cutoff fitness is found, or until a maximum number of generations have transpired.

GAs perform a form of directed *random search*. The efficacy of using this approach relies heavily on the choice of fitness function, the size of each generation, the maximum number of generations, the crossover rate, the mutation rate, and sometimes a scaling factor [6, p. 1–86].

Many variations have been proposed that modify the basic GA paradigm. For example, Rosin and Belew describe [18] a co-evolution method that evolves two separate populations that compete with each other after each generation. The idea is that as population α evolves an individual that can beat individuals in population β, population β will have to evolve in order to keep up. This competition is analogous to different species competing in the wild.

2.2.5 Thread Pools

Though not an AI technique, thread pools are important for our program. Thread pools begin by starting a finite number of threads, each capable of doing some work. When work becomes available, it is given to a thread in the thread pool to do. If there is more work to do than threads available, then the work must simply wait to be done. This method saves some overhead because each time work must be done, the operating system does not need to create and destroy a new thread which can take much time.

A problem with this method is that one must determine an optimal number of threads to start within the pool. The number of processors, the complexity of the problem, and the amount of work to be done all affect the usefulness and size of thread pools.

2.3 AI and Games

The field of artificial intelligence has been applied to the development of game playing programs, particularly two-person games of perfect information. One of the most important ideas is minimax search along with alpha-beta pruning.

2.3.1 Minimax Search

Minimax search involves enumerating all possible moves for one of the two players followed by enumerating every possible response to each of these initial choices. This process is continued until all the leaves of the tree represent final game states. A tree such as this allows the program to play a game perfectly, but for all but the most trivial games this approach requires too much time and memory.

As a compromise, programmers might only allow the tree to expand to a certain level and then assign an estimate of the quality of the game state with an evaluation function. At any given point in the game, the minimax tree allows the programmer to pick the best move assuming the evaluation function is accurate. The problem with this approach is that this evaluation function only approximates the quality of a position and may take quite some time to evaluate. Additionally, if the evaluation function is accurate, then there would be no need to have any tree in the first place.

Alpha-Beta Pruning As an improvement to using an evaluation function and limiting the number of levels, one can use alpha-beta pruning which keeps two values: an α value and a β value. These values store the highest and lowest evaluation values. Low values are good for one side, while high values are good for the other player. Assuming that players always play the best move possible, one can prune tree branches that are worse than some other possible move that the algorithm has seen before. For example, while performing a depth-first search one finds that one player can achieve an evaluation of 10. Then, deeper in the search, a possible sequence of moves leads to a value of five. The subtree with the value five will be pruned and no further moves from that path will be considered. This method always returns the same value as pure minimax search with depth-limitation, so this pruning method is almost always used along with minimax search.

2.3.2 AI and Other Games

Other games of perfect information such as Reversi, Pente, checkers, chess, and go-moku can derive benefits from AI techniques just as go can. The main difference, in our opinion, lies in their branching factors and the manner in which each piece affects other pieces. Reversi, checkers, and chess all have relatively small branching factors making them much more conducive to traditional approaches such as alpha-beta search with move-ordering and other advanced pruning techniques. Pente and go-moku, on the other hand, have similar branching factors to go, but have much simpler interactions between the pieces.

Peter Norvig [13] discusses in depth the construction and refinement of a Lisp program to play Othello. The evaluation function and some of the details are useless for programming go, but the work has much to contribute relative to efficiency issues, searching, and other miscellaneous topics. For example, Norvig uses minimax search with alpha-beta pruning, but he also suggests improvements upon this method. One improvement is to order the moves at each node in the search tree in an attempt to allow pruning to remove more nodes. This ordering can be accomplished if certain locations on the board are more advantageous than others, i.e., better moves are placed first.

Another method is to find the evaluation value for each successor of the current node and then proceed to traverse these, not in an uninformed depth-first manner, but rather in order by evaluation value so that the best successor node is searched next. This potentially allows for a greater number of pruned nodes as well.

Another improvement to playing games of perfect information is to keep track of *killer* moves. This method would have the programmer store moves that were exceedingly bad (moves which were discounted while performing minimax search). If these moves show up later in the search then they are placed first, before other nodes in the minimax tree.

Norvig continues with the idea of generating abstract heuristic values that are relevant to the game such as mobility in the game of Othello or pawn structure in the game of chess. Go, for example, has potential candidates for approaches of this nature such as thickness and good shape which both describe abstract concepts that relate to good moves. One usually wants to build thickness and to make good shape.

Another method is *forward pruning* which requires a function that removes obviously poor moves from the search. It is difficult to do and very subjective. While out of favor as a rigorous method, it is a necessity for games with large branching factors.

Programs that think while the opponent is playing can gain some advantage, and the use of board hashing and opening book databases can help programs' strengths as well. Also, exhaustive searching near the end of a game can be an option for some games such as Othello, but may not be feasible in go.

2.4 AI and Go

2.4.1 Search Space

The number of possible states S on a go board of size 19 is $S = 3^{19^2} \approx 1.74 \times 10^{172}$. There are $19 \times 19 = 19^2$ intersections on the board, and each location has three possible states: black, white, or empty. Though many of the states are very unlikely to occur, one can appreciate the size of the complete search tree. Even accounting for symmetries such as color-inversion symmetry, rotational symmetry, reflection symmetry, and the fact that the number of stones of each color are usually roughly the same, the number is huge.

One of the greatest difficulties in programming go is the immense branching factor in the game. The first move in a game of go can be any one of $19^2 = 361$ moves, while chess has only 20 initial moves. Reversi only has four moves possible at the onset. Though the number of possible moves fluctuates as play progresses, these games cannot be compared to the order of magnitude difference in the branching factor of go. Chess has $20 \times 20 = 400$ game states after the initial two moves, while go has $19^2 \times (19^2 - 1) = 129,960$ game states after the initial two moves! Including a third move brings chess up to approximately $400 \times 25 = 10,000$ states, while go has $129,960 \times (19^2 - 2) = 46,655,640$ states. This example illustrates why nobody has (and possibly ever will) play go well by brute force—there are already three orders of magnitude difference in the size of the search space after just three moves. One can prune many moves throughout the game, but even the ability to prune three-fourths of the moves would result in a huge search space.

2.4.2 Neural Network Techniques

Many programmers and researchers have used neural networks to attempt to play go well including neural networks with GA-evolved weights. For example, Markus Enzenberger [4] created an architecture for his program *NeuroGo* that evaluated the board using a neural network with backpropagation and temporal difference learning. The network received its input from a feature expert, while a relation expert controlled the connections between the layers of the neural network. In addition, there existed an external expert that could override the neural network's output for a small class of problems. The idea of using experts to extract features from the gobans [5] is interesting, but few details of the inner workings of *NeuroGo* were available.

Paul Donnelly *et al.* [3] studied neural networks that were evolved using genetic algorithms. They used a 9×9 board along with a three layer non-recurrent network. They also postulated that recurrent networks with more than a single hidden layer might be better suited for the non-linearities of go. Their experiments consisted of creating a population of 32 networks that all played each other. The

[5] A goban is simply another term for *a board for playing go*.

network winning the most games overwrote the network that lost the most at the end of the cycle. They used the networks to evaluate the quality of a given position which was accomplished via a single output neuron and input neurons that derived their inputs directly from the goban. Each location on the board corresponded to three individual input neurons (one each for white, black, and empty positions). The resulting input layer thus had $9 \times 9 \times 3 = 243$ neurons. The authors found that the networks slowly got better, but the network still played poorly compared to modern go programs. This approach theoretically has merit, but to implement this architecture on a full board using only two hidden layers each analogous to those in the paper, one would need $19 \times 19 \times 3 = 1083$ input neurons and $19 \times 19 = 361$ neurons in each hidden layer. This results in $1083 \times 361 + 361 \times 361 + 361 = 521,645$ connections. A network of this caliber could be constructed, but recurrent connections might be required for it to play well, and training time would be prohibitive.

2.4.3 Traditional Techniques in Go Programs

Some go programs do not use any soft computing techniques, i.e., they do not rely on learning, genetic algorithms, neural networks, cellular automatons, or other similar approaches.

In *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory* [10], Müller studied methods of playing go that generate moves by first enumerating possible moves based on small, local views of the goban. These moves are filtered, ordered, checked, and refiltered. The best move is executed. If a ko ensues, a special ko module is called. If no move survives this process, the program passes. At the core of this approach is a pattern matching database that uses Patricia trees which is a method normally used to search large text databases such as dictionaries. This program contained about 3000 patterns, and pattern matching was its chief element. This reflects a very prominent trend across many go programs: they often rely heavily on vast databases of patterns that have been built by hand. These pattern databases make these programs better, and the implementation of these databases is not a trivial task. The use of large databases proves nothing about a program's "intelligence" since it becomes in essence a sophisticated lookup table. There does remain the possibility of learning patterns as the program plays, but techniques such as these would not fall under the category of traditional techniques.

Another prominent program, *The Many Faces of Go* [5], as of 1993 had an opening move database that contained around 45,000 moves and a pattern database of about 1,000 patterns. This program contained a rule-based expert system with around 200 rules that were used to suggest moves to look into further. Additionally, dynamic knowledge was stored about the state of the board which was generated with algorithmic C-code [5].

Though this investigation of traditional techniques has been very cursory, these programs represent some prominent themes in most strong go programs: they construct meta-data based on the state of the board and use this meta-data along with large databases of patterns to decide what move to play next. Rarely does learning or *extensive* minimax-style search play a role in the skill of these games.

2.4.4 Genetic Algorithm Techniques in Go Programs

Many attempts have been made to create a program that plays go by using genetic algorithms. None have been successful at creating a world-class player, but nobody has accomplished this feat without genetic algorithms either. What follows is a perusal of some attempts to use genetic algorithms to play this game.

Da Silva [2] used GAs to evolve a go evaluation function for 7×7 boards. The evaluation function worked by attempting to translate a given board into a new board that represented how the final configuration of the game would be. The evaluation function then looked at who *won* to calculate the fitness. Essentially, the genetic algorithm attempted to evolve an evaluation function that could be used in minimax searches with alpha-beta pruning. The evolved parameters were a set of low-level functions that performed simple calculations based on the board state. These functions, organized as the chromosome dictates, produced what the author called an *S-expression*, which is a significant component of calculating a board evaluation and consequently the fitness. Da Silva's approach yielded a player that on average never beat a defacto opponent called *Wally*, a freely available public domain go program.

Jeffrey Greenberg has written a program using genetic algorithms to play go [7]. He feels that go represents a good test-bed that approaches the complexity of real problems while not being as complex as a commercial application. One could argue with this premise since go is easily as complex to program as a modern commercial software package—why else would modern go programs remain such poor players? Aside from this point of view, he wrote a GA engine in C++ independent of go. Knowledge in this program is represented entirely by triples reminiscent of Prolog predicates such as *IfPointAt(x, y, z)*. These statements can be nested. Detailed descriptions were scant, but it appears that each variable (x, y, or z) is comprised of a board location, the color (white, black, or empty), and the action to take (move, pass, or resign). If the parameter x is satisfied, then y is checked, otherwise z is checked. Through this possibly layered traversal of the statements, moves are chosen. The program, "…was very poor at breeding individuals that could match. And when it did, the individual would often resign after but a few moves" [7].

In [9], the researchers used genetic programming and the game of go to create genetic algorithms that incorporate qualities of true human experts. One inclusion was to incorporate useful but infrequently used rules, and another was to model ecological systems. The ecological models dictate that many species coexist. Their ideas revolve around the fact that species live together in an environment, yet they can be radically different. Rules, in their system, increase in number and eat virtual *food*. Rules whose activations decrease to zero, die, while rules whose activations become too high split into the original rule and a more specific rule. A training datum is considered food, which is eaten by a rule that matches it; the activation value of the rule then increases. These researchers used their genetic algorithm entirely to evolve rules based on patterns found on the board. The authors did not report the playing skill of their program, but they did present the rules that the program generated to go experts. These experts decided that 41.6% were good, 21.1% were average, and 37.3% were bad [9].

2.4.5 Other Techniques and Hybrids

In [17], the authors describe their *SANE* architecture that evolves neural networks to play go by using genetic algorithms. The program starts with no prior go knowledge at all. The process involves evolving individual neurons using crossover mutations and random point mutations. Each neuron is defined as a set of bits that describe connections and the connection weights. Each neuron has a fixed number of connections, but each connection can be attached to either the output or the input layer. Network blueprints are also evolved along with the individual neurons. Entire networks are evolved based on the final state of the game rather than assigning credit to individual moves, which the authors state is unreasonable; however, it could be argued that one can simply assume that game records between two masters represent on average the best move at each point in the game. This may not actually be true, but it is a close enough approximation.

In [14], the researchers discuss the evolution of neural networks on a variant of the SANE architecture that evolves individual neurons, but evaluates the fitness of entire networks. In addition, blueprints (i.e., sets of neurons that work well together) are evolved. The neurons in question are only for the single hidden layer of the network. SANE has been shown to work well in continuous domains and games with hidden state information. The authors describe their EuSANE architecture:

> "The core idea of EuA is to select every allele of the offspring separately, based on explicit analysis of the allele fitness distributions in the population. It furthermore contains a restriction operator that focuses the analysis on members of the population most relevant for determining the next allele. In every generation only one new individual is generated, implementing a steady-state replacement." [14]

Chapter 3

Methodology

Our approach consists of a three layer summation network with each layer fully connected to its adjacent layers. Each connection is characterized by an integer weight, and each node sums arrays. These arrays each contain an element that corresponds to locations on the board (i.e., it is a one-to-one mapping). The cornerstone of our design is to evolve these weights using our genetic algorithm, thus each chromosome specifies a set of integer weights for the summation network. The initial inputs to the network are the outputs from the individual agents.

3.1 Design Overview

Exodus, as the program we wrote is called, provides the end user with the ability to run regressions, evolve a GA player using stored game training sets, and play a human player with extensibility in mind to allow IGS [1] and gomodem[2] connectivity in the future.

Exodus was designed in a highly object-oriented fashion as will be described in this chapter. It consists of a moderator that allows two move generation classes (called interfaces) to play against each other. Through this abstract interface class, we have developed an ASCII text player that interfaces with a human user, a simple Perl/Tk[3] interface that also interfaces with a human user, a GA player that will be described in detail below, and a GA trainer that is designed to play against the GA player in order to calculate the fitness of the GA player. The interface's simplicity allows for the potential future development of interfaces that can play go over the Internet or interfaces that communicate over a serial line, i.e., as used in competitions.

3.2 Stone, Board, and Game Classes

The stone class represents a single location on the goban, which was implemented with speed as the primary concern. It uses bit operations to test various traits of a location such as if the location has a black stone or a white stone. This feature eliminates many potential modulo operations that would be necessary otherwise.

[1] Internet Go Server

[2] A protocol for serial communication between two computers, each playing go.

[3] Tk is a graphical package, originally implemented for use by the language TCL, that provides basic graphic capabilities such as window creation, buttons, frames, text boxes, etc.

	A	B	C	D	E	F	G	H	J	
9	0	1	2	3	4	5	6	7	8	0
8	9	10	11	12	13	14	15	16	17	1
7	18	19	20	21	22	23	24	25	26	2
6	27	28	29	30	31	32	33	34	35	3
5	36	37	38	39	40	41	42	43	44	4
4	45	46	47	48	49	50	51	52	53	5
3	54	55	56	57	58	59	60	61	62	6
2	63	64	65	66	67	68	69	70	71	7
1	72	73	74	75	76	77	78	79	80	8
	0	1	2	3	4	5	6	7	8	

Figure 3.1: Board Locations For a 9×9 Board.

It also has functions that test if the location is on an edge. This class can be found in section A.2.29.

The next layer of abstraction encapsulates the concept of a board, which is simply a one-dimensional array of stones. A one-dimensional array was chosen in an attempt to speed up board manipulations by reducing the need for pointer arithmetic that is required in multiarray offset calculations. Figure 3.1 shows how a 9×9 board is represented and shows how the two-dimensional structure is mapped onto a one-dimensional array. Stones on the edges are marked as such to allow tests such as *stone[9].left()* or *stone[32].nottop()*.

On top of the board abstraction there is a game class which stores a linked list of boards and keeps track of which side's turn it is. The game class enforces certain optional rules such as whether or not to allow suicide. It also provides functions such as *play_move* and *legal*, both of which have obvious uses. The source code for this class is found in section A.2.9.

```
   _A_B_C_D_E_F_G_H_J_
9| . . . . # # # . . | 9
8| # # # # # o o o o | 8
7| o o o o o o o o o | 7
6| o # # o . o . . . | 6
5| # . # . . . . # . | 5
4| . . # . . . . . . | 4
3| . . . . # . . . . | 3
2| . . . . . . . # . | 2
1| . . . . . . . . . | 1
   _A_B_C_D_E_F_G_H_J_
```

Figure 3.2: An ASCII Board.

3.3 User Interfaces

The Exodus program contains two distinct user interfaces, each of which inherits from a superclass *Interface* found in section A.2.18: a text interface and and a graphical interface. The text interface displays the goban using ASCII characters with a `#` representing black and an `o` representing white. Figure 3.2 shows an ASCII board for a 9×9 game. This interface is useful when visual appeal is not an issue (i.e. testing code, not directly related to the output of the board).

Another user interface is a GUI interface that uses an external Perl/Tk program to display the goban. Figure 3.3 shows a screen shot. This interface is important for playing games against the program (a graphical board is easier to interact with). This interface was also useful while developing the board and game classes as it made debugging easier. A graphical user interface allowed for a quicker way to *play* with the program in an attempt to find problems or bugs.

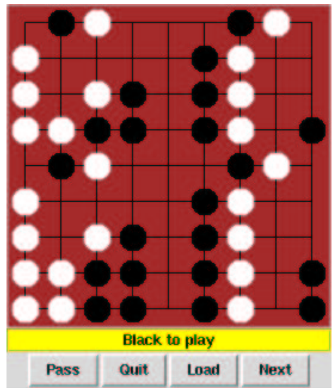


Figure 3.3: Graphical User Interface Screen Shot.

3.4 Genetic Algorithm

The code for performing genetic algorithms was originally taken from David E. Goldberg [6]. The code in this text was converted to C++ and syntactically modified to better suit an object-oriented approach like ours. The code can be found in section A.2.8.

The GA code was made as generic as possible and supplies a member function called *set_codex* which allows the reception of a pointer to a class of type *PreCodex*, which is a superclass of any class that wishes to supply a fitness function.

The program keeps statistics on the performance of the GA and tracts the minimum, maximum, sum, average, variance, and standard deviation of the fitnesses from each generation. In addition to these, F-test and T-test values are computed for each generation, comparing the statistics to the initial generation. The F-test value calculates whether two distributions have significantly different variances. The T-test (Student's T-test) measures whether two distributions have significantly different means. Two versions of the T-test were used. One version is used for distributions with statistically different variances and the other for distributions with statistically identical variances. These numbers allow us to better determine the significance of the data.

3.5 Moderator

The moderator class, found in section A.2.19, essentially loads two move generators which can manifest themselves as anything from a user interface to a random move generator or a genetic algorithm player. There also exists a genetic algorithm trainer which is described in section 3.7.

The moderator class is multi-threaded, allowing a thread for each move generator. This design allows both sides to have processing time throughout the entire game—not just during a side's turn. Another feature of this class is that it was implemented as a template, which allows the two players to be specified when one instantiates a moderator class. Message passing is used to allow communication between the moderator and the two move generators.

3.5.1 Probability Board

The probability board class, found in section A.2.27, is a conceptually simple abstraction that stores an array of values which correspond with the locations on the goban. The semantics are such that the values at each element represent how highly that location is valued as a possible next move. Each agent constructs one of these, and to facilitate the aforementioned use of this class, a *spin* function was implemented (to choose a move probabilistically), and a *normalize* function was implemented to facilitate the addition of two or more of the boards together, each from a potentially difference source. Also, the ability to multiply each board by a scalar value was added (whose use will become apparent in section 3.6).

3.6 Agent Network Architecture

The GA player uses a thread pool to run multiple agents that each generate a numerical value for each board location. These arrays are multiplied by GA-evolved weights, added together, normalized, and fed through a second layer of summation nodes. The resulting array is then normalized. The highest value in this result array then becomes the move played. Figure 1.1 shows a graphical representation of this process which is described algorithmically as follows:

1. Each of N agents computes a value for each location on the goban. This *probability board* is a vector and shall be denoted as β_n where n is the agent number.

2. Each of the K second level nodes γ_k sum all β_n values multiplied by a scalar value $w_{k,n}$. Thus,

$$\gamma_k = \sum_{n=0}^{N} w_{k,n} \cdot \beta_n$$

3. These γ_k are vectors that are then normalized so that the values add up to 1 in each vector unless all of the values in a vector are zero, in which case they are left that way.

4. These normalized γ-vectors are then multiplied by a second set of weights and added together:

$$\epsilon = \sum_{k=0}^{K} w_k \cdot \gamma_k$$

5. This final vector, ϵ, is normalized and represents a distribution of which move to play. To make training the GA simpler, we simply choose the first highest value rather than choosing the move to play probabilistically, though either way is possible.

This approach theoretically allows for a large number of agents, limited primarily by the size of the thread pool and the number of processors available to the program. A major goal of this project was to create a design that was scalable and could benefit from a highly parallel machine. Though scalability was not tested, the possibility of adding more agents could easily be realized. Figuring out what each agent would do could become a significant bottleneck, though.

3.7 Genetic Algorithm Trainer

This move generator was designed to play against the genetic algorithm player. It reads a sequence of moves from a data file (which were derived from recorded games of professionals in the public domain). It sets up the board and then allows the genetic algorithm player to play. After the GA player has played, the trainer resets the game state to whatever the professional actually played in the game record. The colors on the board are flipped, and the GA player is allowed to play again. The colors are flipped to allow the GA player, which plays a single color, to gain benefit from the entire game record and not just from the plays of a single color. After all, the recorded games are from two professionals playing, and each player can be assumed to be playing well. The usefulness of this GA trainer player, which is shown in section A.2.10, will become apparent in section 3.8.2, which describes the fitness function in detail.

3.8 Genetic Algorithm Player

3.8.1 GA Player Details

This code, found in section A.2.11, loads the parameters for the weights in the summation network (described in section 3.6) and computes the move to play by running the agents, filtering their values through the summation network, and then picking either the first highest value or normalizing and then choosing the move probabilistically. This class also inherits from *PreCodex* which implies that it provides a fitness function (that the GA uses).

3.8.2 Fitness Function

The fitness is calculated by setting up a goban as dictated by stored games from the Internet. If the GA player chooses the correct next move, an accumulator is incremented. The fitness is then simply the percentage of moves correctly played. Many other GA go programs calculate fitness by using some variation of attempting to guess how the current board configuration relates to the final division of points at the end of the game. Our approach sidesteps this difficulty which relates closely with the difficulty of simply scoring a finished game. The fitness function code is shown in section A.2.11.

3.9 Agents

We have designed and implemented six different agents that each choose moves in significantly different ways. Currently, there is a random agent that plays random legal moves, a follower agent that tries to play close to the enemy, an opener agent that plays in the locations usually played in at the beginning of a game, a capture agent that attempts to kill groups by reducing other groups' liberties, an agent that attempts to create a strong configuration known as a tiger's mouth, and an extension agent that favors moves close to friendly stones.

3.9.1 Random Agent

An agent that plays random legal moves was developed to allow the testing of code that directly uses the agents and to allow the testing of the code that lets the agents interact. Additionally, the random agent was used as a baseline with which the other agents can compare themselves. For example, the standard by which the success of the genetic algorithms is judged is the set of five random agents. Section A.2.28 contains the code for the random agent.

3.9.2 Follower Agent

The follower agent, found in section A.2.7, values playing on locations adjacent to enemy stones. As is often found in games of go, many good moves are often near enemy stones, i.e., attacking them. Playing close to enemy stones not only attacks them, but also attempts to push the enemy group in the opposite direction.

3.9.3 Opener Agent

This agent, found in section A.2.24, suggests moves around the perimeter of the board near the third or fourth row. The values decrease the further the game progresses. The reasoning behind this type of agent is that at the very beginning of most games, stones are played near the edges and sides because this is where it is easiest to make territory. In a corner, one only has to worry about attacks from two directions. On a side, attacks are only possible from three directions, while in the middle, attacks can be made from all directions. These considerations are what justifies having an opener agent.

3.9.4 Capturer Agent

This agent attempts to capture enemy stones by filling in their last liberties. It has no knowledge of living or dead groups, thus it plays simply by calculating which groups have one or two liberties left and then plays in those liberties. Located in section A.2.14 and called *GroupStatsAgent*, this agent does not take into account moves that would reduce a friendly group's liberty count down to one. What this means is that this agent would be perfectly content to play a move that reduced an enemy's group to a single liberty while that very same move would allow the

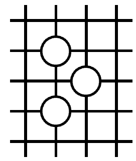


Figure 3.4: Tiger's Mouth Formation.

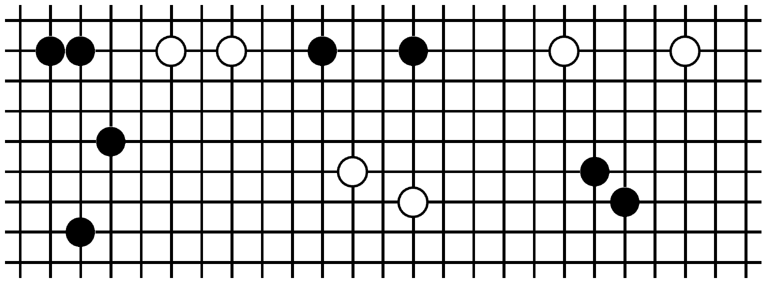
enemy to capture a friendly group on the next turn.

3.9.5 Tiger's Mouth Agent

The tiger's mouth agent (section A.2.34) attempts to create a powerful configuration called a *tiger's mouth*. Figure 3.4 shows what a tiger's mouth looks like. This formation retains the same name regardless of all symmetries. This configuration is considered strong because it allows three stones to NOT be connected while retaining the ability to become connected by playing in the center location. Another strength of this configuration is that if an enemy stone tries to keep these stones from connecting, that enemy stone can be captured on the next turn if it is not part of another group. The versatility of this formation provides justification for the inclusion of this agent.

3.9.6 Extender Agent

This agent plays many of the common extensions. Each type of extension has a different weight (or value) which is derived from the GA chromosome. It is also



Extensions starting at the upper-left and continuing clockwise. 1. Extension • 2. One-point extension • 3. Two-point extension • 4. Three-point extension • 5. Shoulder extension • 6. Knight's move • 7. Large Knight's move

Figure 3.5: Extensions.

the only agent that uses alleles from the chromosomes to set these internal values (in this case, the alleles specify the relative value of each of the extension types). These types of extensions are shown in Figure 3.5.

3.10 Regressions

We have written extensive amounts of code to test the validity and accuracy of much of the code. Nearly every function in every class has some sort of regression. The regressions for each source file are located within that file. One cannot guarantee program correctness by running the regressions, but the regressions do serve to instill a greater feeling of security that incremental changes to the code do not break anything coded previously.

3.11 Unimplemented Features

Over the course of designing and implementing this program many ideas that were designed into the original program were not actually implemented, though integration of these parts would be relatively simple given enough time. These unimplemented features include:

- The ability to have the program play against real people on IGS (Internet Go Server).
- A blackboard architecture for agent communication.
- The ability to play another program via a protocol called the *gomodem protocol*.
- The ability to track time during games.
- The modification of agents to allow the use of time when the opponent is thinking to do useful work.
- The incorporation of interagent communication.
- The ability to score finished games.

Chapter 4

Experiments and Results

The experiment descriptions and results that follow attempt to show the successful evolution of summation network weights for a multiagent approach to playing go. The key point is that we want to illustrate that though each individual agent may play poorly, the agents playing together actually play better. We wish to further show that random search (using a GA), finds weights for the summation network that improve over multiple generations.

The fitness function for our experiments uses recorded games; we used recorded 9×9 games between professionals from the public domain that occurred between 1995 and 2000 on an international go server [1].

4.1 Individual Agent Experiments

The genetic algorithm was run for eight generations with the program configured to use only a single agent. These agents were the random-move-generating agent, the extension agent, the follower agent, the capture agent, the opening move agent, and the tiger's eye agent. In each case, the populations contained ten individuals. Such a small population and small number of generations were used because of the large amount of time it took to run the GAs even with this configuration. These runs took around three days on a dual-processor, 1.2GHz machine. Additionally, most of the single-agent configurations do not benefit from the GA on their own, making the time required for a larger population or a larger number of generations of questionable use. The crossover percentage was 40% with a mutation probability of 0.0333. Additionally, the F-multiplier was two. The random agent was used as a baseline. After evolution by the genetic algorithm, the best individual in the final population was used to play against a testing data-set representing game records that were different than those used to train during the run of the genetic algorithm.

4.1.1 Opener Agent

Table 4.1 shows the results of the genetic algorithm run using only the opener agent. These data (fitness values) are also shown in Figure 4.1. Since the genetic

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.0114	0.0114	0.0114	0	0.114
1	0.0114	0.0114	0.0114	0	0.114
2	0.0114	0.0114	0.0114	0	0.114
3	0.0114	0.0114	0.0114	0	0.114
4	0.0114	0.0114	0.0114	0	0.114
5	0.0114	0.0114	0.0114	0	0.114
6	0.0114	0.0114	0.0114	0	0.114
7	0.0114	0.0114	0.0114	0	0.114
8	0.0114	0.0114	0.0114	0	0.114

Table 4.1: Opener Agent Data.

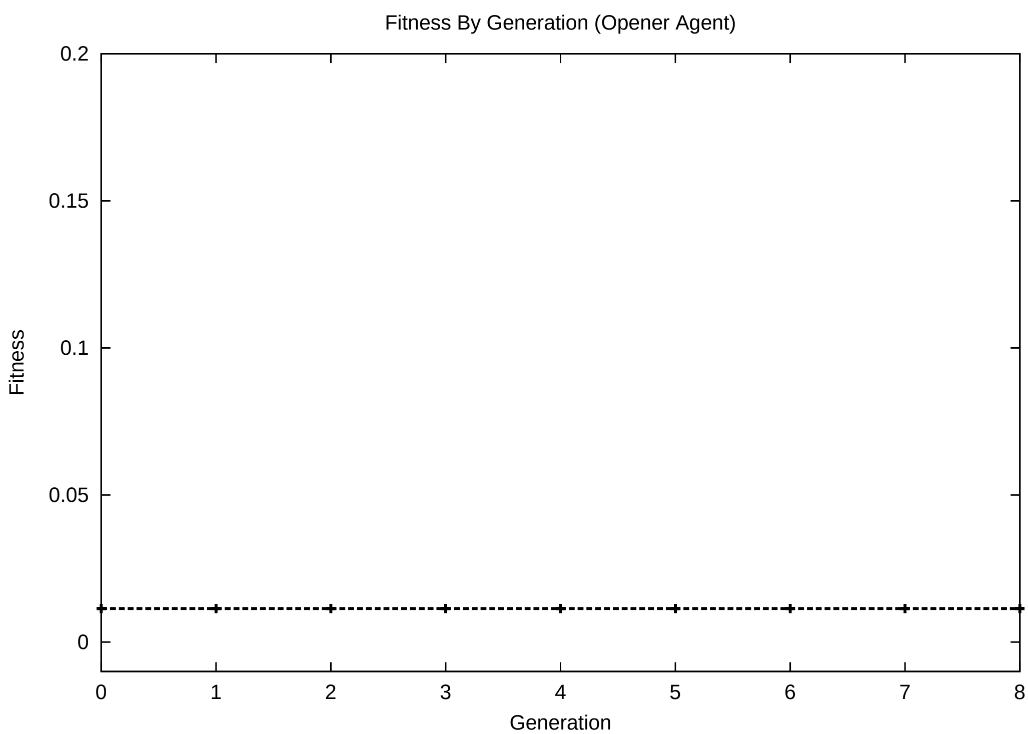


Figure 4.1: GA Data Plot With Opener Agent.

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.00622	0.00622	0.00622	4.91e-10	0.0622
1	0.00622	0.00622	0.00622	7.39e-06	0.0622
2	0.00622	0.00622	0.00622	7.39e-06	0.0622
3	0.00622	0.00622	0.00622	0.000906	0.0622
4	0.00622	0.00622	0.00622	0.000906	0.0622
5	0.00622	0.00622	0.00622	0.01	0.0622
6	0.00622	0.00622	0.00622	0.01	0.0622
7	0.00622	0.00622	0.00622	0.0334	0.0622
8	0.00622	0.00622	0.00622	0.0334	0.0622

Table 4.2: Randomly Playing Agent Data.

algorithm-evolved weights are not effective if a single agent is used, one would expect that the fitness values would not change, which is exactly what appears to have happened here. The best chromosome (which incidentally is arbitrary) of the last generation chose 1.14% of the training moves correctly and 1.55% of the testing moves correctly. Considering that this agent was designed to play opening moves, this is not a surprise that it fared so poorly. The F-test and T-test (described in section 3.4) have little use here in a straight-forward example such as this. Each distribution of each generation is clearly identical to each other, so nothing was gained from the genetic algorithm.

4.1.2 Single Random Agent

The single randomly playing agent did not fare well as shown in the data (Table 4.2 and Figure 4.2). Since the random agents always at least pick legal moves, the number of possible moves near the end of any game becomes smaller, which increases the likelihood that a random guess would be correct. These considerations

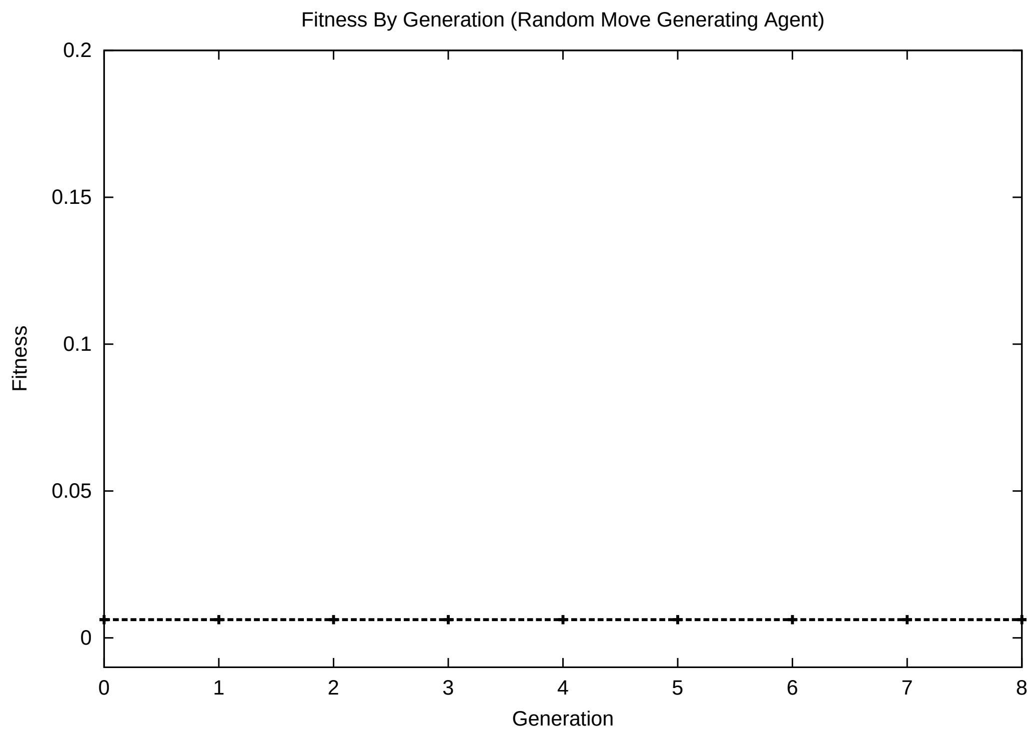


Figure 4.2: GA Data Plot With Randomly Playing Agent.

aside, one should note that the random agents are not actually randomly choosing locations to play, but rather assigning the same value for every legal position to move to. The final resulting probability board (see section 3.5.1 above) contains an array of values (which in this case would all be the same). The program can be configured to either pick the first highest or to pick one probabilistically. For this experiment (and all of the others as well), the first, more deterministic path was taken. The result of this is that the first legal move is always chosen, which ends up being correct a static number of times. We hypothesize that the 0.622% of the moves that the best chromosome of the eighth generation got correct was a result of this effect. If one keeps choosing the same legal location as one's guess, it

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.0415	0.0301	0.0365	0.00428	0.365
1	0.0689	0	0.0354	0.0319	0.354
2	0.0777	0.0037	0.0389	0.0347	0.389
3	0.0788	0.00656	0.0394	0.0641	0.394
4	0.0753	2.76e-10	0.0409	0.0667	0.409
5	0.0821	8.1e-09	0.0436	0.0899	0.436
6	0.0797	5.57e-09	0.0458	0.0893	0.458
7	0.0669	0	0.0468	0.102	0.468
8	0.0861	0	0.0449	0.105	0.449

Table 4.3: Extension Agent Data.

eventually becomes correct. An interesting feature of these data are that the testing data yielded 0.62% correct moves which is not surprising given the previous explanation.

4.1.3 Extension Agent

The extension agent is more interesting, in that there are internal parameters to the agent that derive their values from the evolved chromosomes. Table 4.3 shows the results of the genetic algorithm run using only the extension agent, and the data are also shown graphically in Figure 4.3. The genetic algorithm-evolved weights still do not matter for this single agent, but this agent has internal parameters that could benefit from evolution. As one would hope, as the generations progressed, the mean fitness and the maximum fitness rose. Additionally, the minimum fitness had a net decrease of 0.0301 by the end of the eighth generation. To back up these observations, the F-test predicts that the first and the final generations have insignificantly different variances which allowed us to use

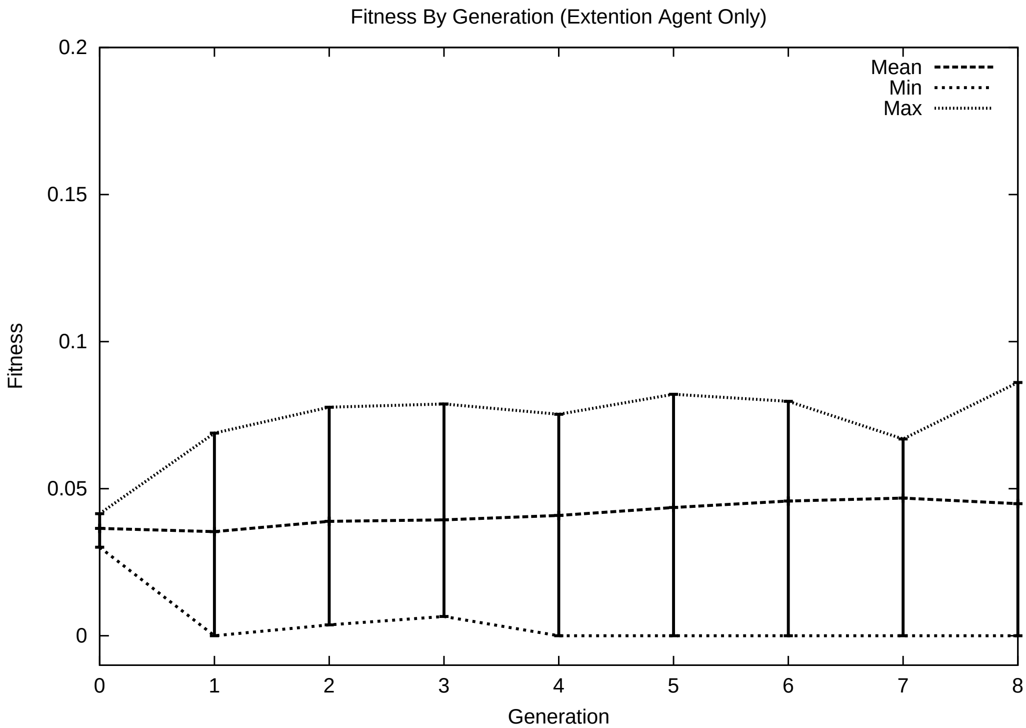


Figure 4.3: GA Data Plot With Extension Agent.

the T-test to predict with a probability of 99.9642% that the improvement is real and not a result of chance. The decrease of the minimum is not a concern due to the increase of the maximum and the mean. The best chromosome of the last generation chose 5.18% of the training moves correctly and 3.88% of the testing moves correctly.

4.1.4 Capturer Agent

The data for this agent (shown in Table 4.4 and graphically in Figure 4.4) shows the same lack of improvement as other individual agents because this agent has no internal parameters that might benefit from evolution. An interesting feature,

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.0777	0.0777	0.0777	7.85e-09	0.777
1	0.0777	0.0777	0.0777	2.95e-05	0.777
2	0.0777	0.0777	0.0777	2.95e-05	0.777
3	0.0777	0.0777	0.0777	0.00181	0.777
4	0.0777	0.0777	0.0777	0.00181	0.777
5	0.0777	0.0777	0.0777	0.0142	0.777
6	0.0777	0.0777	0.0777	0.0142	0.777
7	0.0777	0.0777	0.0777	0.0397	0.777
8	0.0777	0.0777	0.0777	0.0397	0.777

Table 4.4: Capturer Agent Data.

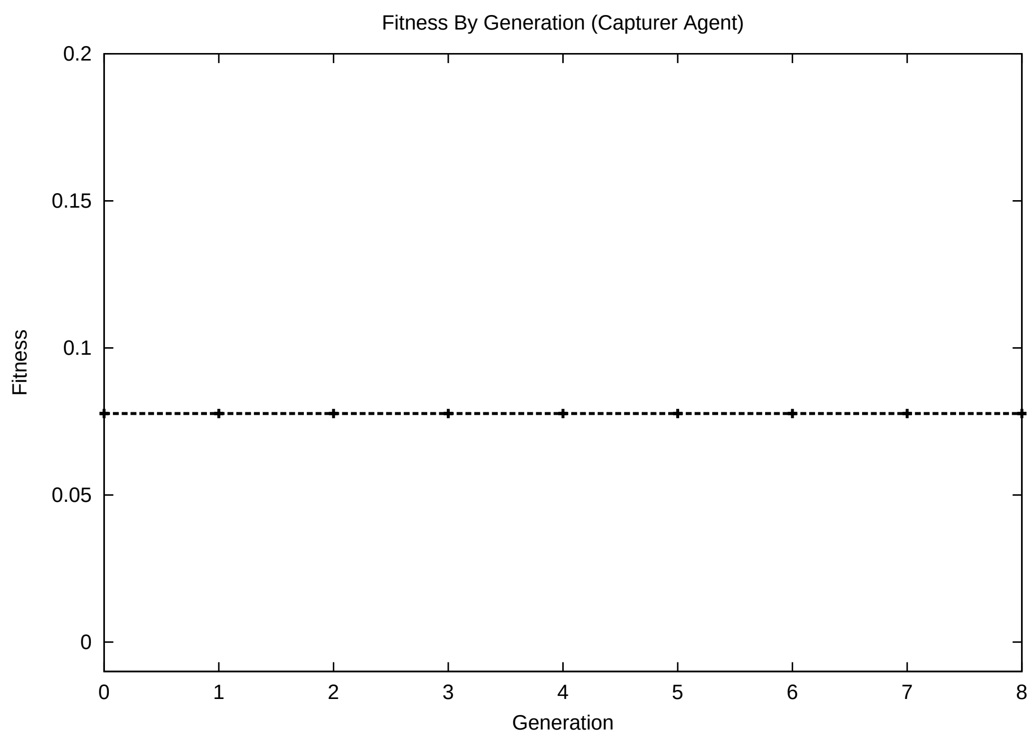


Figure 4.4: GA Data Plot With Capturer Agent.

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.0415	0.0415	0.0415	3.93e-09	0.415
1	0.0415	0.0415	0.0415	2.09e-05	0.415
2	0.0415	0.0415	0.0415	2.09e-05	0.415
3	0.0415	0.0415	0.0415	0.00152	0.415
4	0.0415	0.0415	0.0415	0.00152	0.415
5	0.0415	0.0415	0.0415	0.013	0.415
6	0.0415	0.0415	0.0415	0.013	0.415
7	0.0415	0.0415	0.0415	0.038	0.415
8	0.0415	0.0415	0.0415	0.038	0.415

Table 4.5: Follower Agent Data.

though, is the change in standard deviation, which probably resulted from numerical round-off errors that resulted from the summation network calculations. The best chromosome from the last generation got 7.77% of the moves correct and 4.03% of the testing moves correct.

4.1.5 Follower Agent

The follower agent followed in the footsteps of the other single agents with its lack of improvement. No internal parameters for the genetic algorithm were used. The best chromosome of the last generation got 4.15% of the training moves correct, while it got 3.26% of the testing moves correct. The data can be found in Table 4.5 and in Figure 4.5.

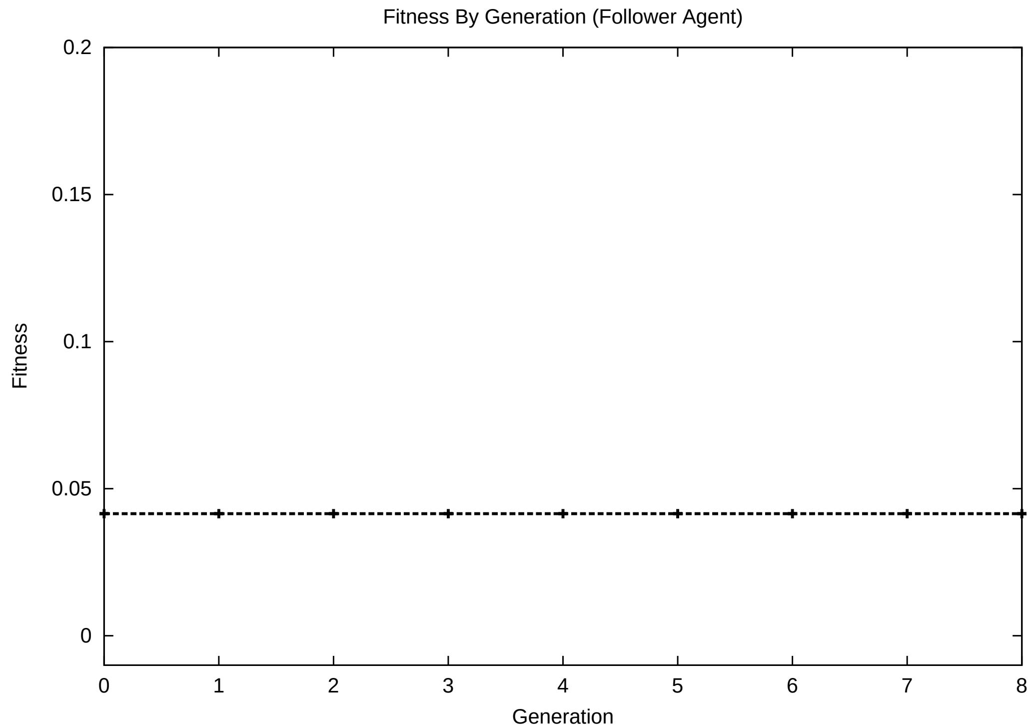


Figure 4.5: GA Data Plot With Follower Agent.

4.1.6 Tiger's Mouth Agent

The last single-agent configuration's data are shown in Table 4.6 and in Figure 4.6. The tiger's eye agent showed no improvement due to a lack of internal genetic algorithm parameters. The best chromosome from the final generation got 2.38% of the moves correct, while it got 2.17% of the moves correct on the testing data.

4.2 Multiagent Experiments

The multiagent experiments closely mirrored the individual agent experiments with the exception that in these cases the program was run with all of the agents

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.0238	0.0238	0.0238	0	0.238
1	0.0238	0.0238	0.0238	0	0.238
2	0.0238	0.0238	0.0238	0	0.238
3	0.0238	0.0238	0.0238	0	0.238
4	0.0238	0.0238	0.0238	0	0.238
5	0.0238	0.0238	0.0238	0	0.238
6	0.0238	0.0238	0.0238	0	0.238
7	0.0238	0.0238	0.0238	0	0.238
8	0.0238	0.0238	0.0238	0	0.238

Table 4.6: Tiger's Mouth Agent Data.

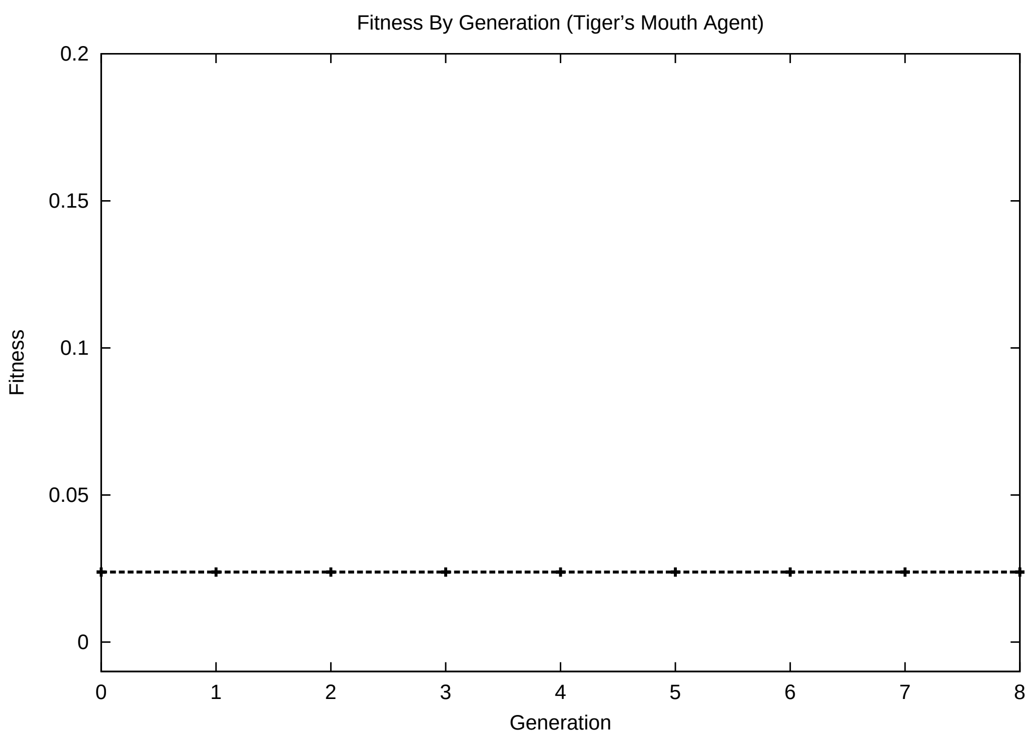


Figure 4.6: GA Data Plot With Tiger's Mouth Agent.

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.00622	0.00622	0.00622	4.91e-10	0.0622
1	0.00622	0.00622	0.00622	7.39e-06	0.0622
2	0.00622	0.00622	0.00622	7.39e-06	0.0622
3	0.00622	0.00622	0.00622	0.000906	0.0622
4	0.00622	0.00622	0.00622	0.000906	0.0622
5	0.00622	0.00622	0.00622	0.01	0.0622
6	0.00622	0.00622	0.00622	0.01	0.0622
7	0.00622	0.00622	0.00622	0.0334	0.0622
8	0.00622	0.00622	0.00622	0.0334	0.0622

Table 4.7: Five Random Agent Data.

at once excluding the random-move-generating agent. A separate run that used five random-move-generating agents was used as a baseline.

4.2.1 Five Random Agents

Not surprisingly, the genetic algorithm configured with five identical random legal move generating agents performed rather poorly. The results were nearly identical to those of the single random agent above. The results are shown in Table 4.7 and in Figure 4.7.

4.2.2 Multiagent Configuration

Table 4.8 and Figure 4.8 show the results of evolving the genetic algorithm using five agents: Opener, Extension, GroupStats, Follower, and TigersMouth agents. Three hidden-layer nodes were used, and each generation had 10 individuals. Initially, the maximum fitness was 0.0881 and the mean fitness was 0.0537. By

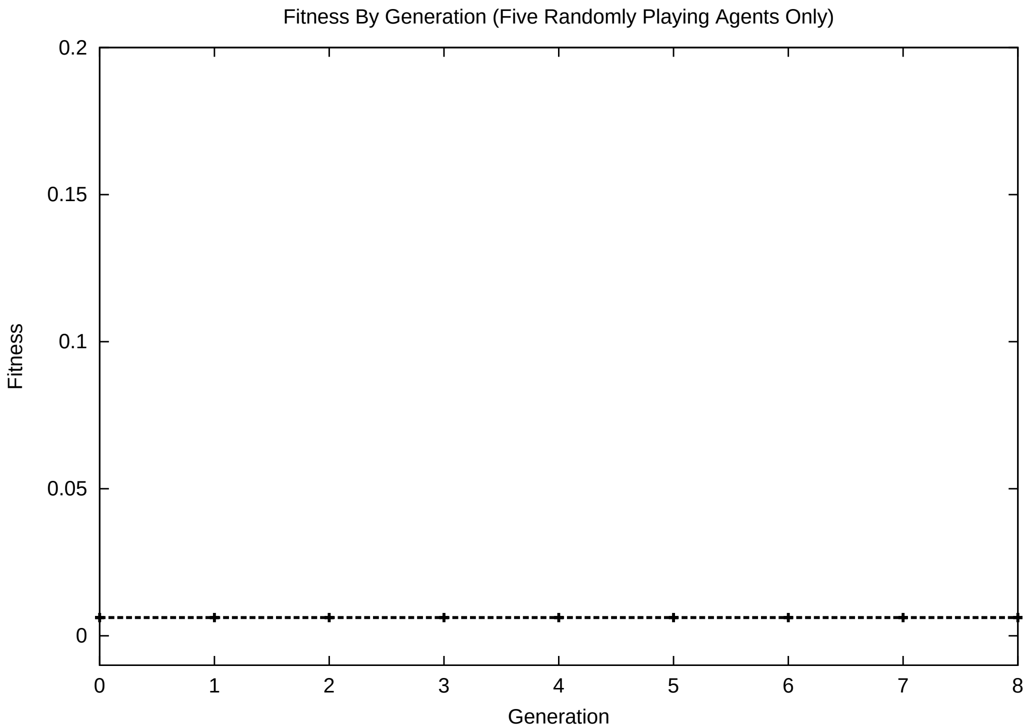


Figure 4.7: GA Data Plot With Five Random Agents.

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.0881	0.0435	0.0537	0.0138	0.537
1	0.108	0.0245	0.0539	0.0454	0.539
2	0.121	0.0106	0.0604	0.0575	0.604
3	0.119	3.49e-10	0.0694	0.0865	0.694
4	0.105	3.15e-09	0.0812	0.0867	0.812
5	0.109	2.82e-09	0.0848	0.103	0.848
6	0.108	2.15e-09	0.0877	0.103	0.877
7	0.134	0	0.087	0.113	0.87
8	0.14	0	0.0798	0.118	0.798

Table 4.8: All Five Agents Data.

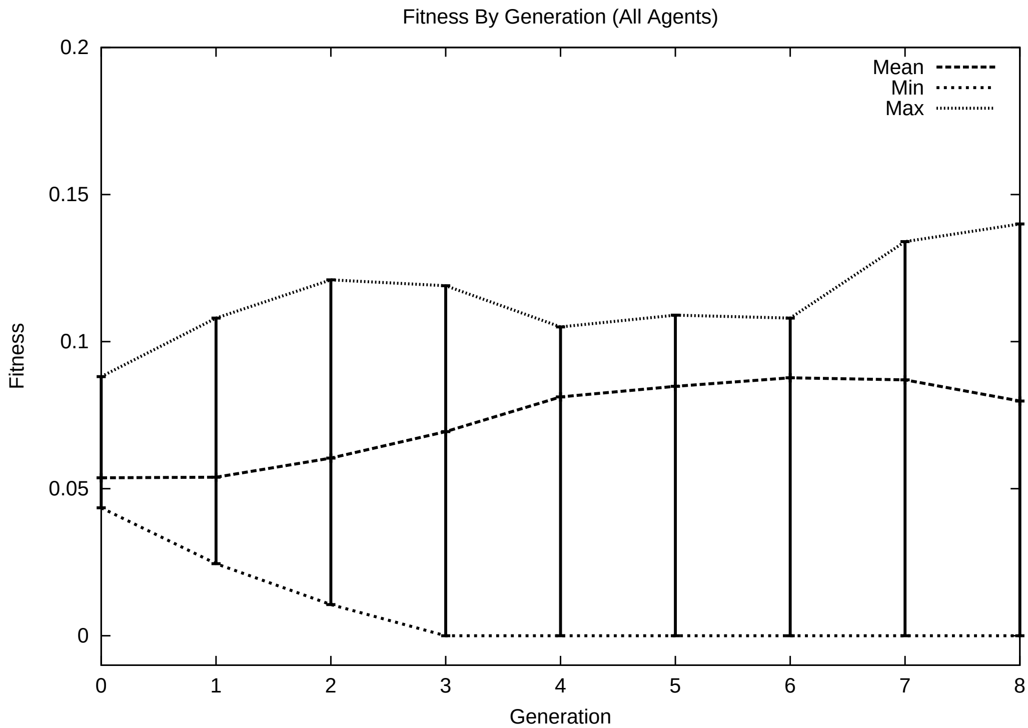


Figure 4.8: GA Data Plot With All Agents.

the final generation, the maximum fitness had risen to 0.14 and the mean fitness had risen to 0.0798. The question then becomes one of deciding if this difference should be attributed to chance or to legitimate improvement. Using the F-test, the difference in the variances was not significant. The T-test value of the final generation was -4.23 which implied a probability of 0.000504 that these results were from chance and not from a different population as the initial population, i.e., the confidence interval was 99.95% that the difference in the means was significant. The best chromosome from the final generation got 10.2% of the moves correct while it got 5.558% of the testing set correct.

The agents were loaded in the following order: OpenerAgent, TigersMouthAgent, GroupStatsAgent (capturer), FollowerAgent, and ExtenderAgent. The final best network configuration had weights from the agents to the second layer of the network as...

$$Weights = \begin{pmatrix} 12 & 15 & 13 \\ 11 & 8 & 15 \\ 10 & 15 & 14 \\ 10 & 3 & 1 \\ 5 & 0 & 13 \end{pmatrix}$$

where each row corresponds to an agent and each column corresponds to a node in the next layer. The weights from this next layer to the output node is...

$$\begin{pmatrix} 11 \\ 15 \\ 3 \end{pmatrix}$$

4.2.3 Multiagent Configuration, Large Population

Table 4.9 and Figure 4.9 show the results after seven generations of the multiagent configuration with a population size of 100. All parameters were the same as the smaller multiagent configuration except for the population size. These data support the results from the smaller multiagent experiment.

Generation	Max	Min	Mean	Std. Dev.	Sumfitness
0	0.0995	0.0321	0.0549	0.0142	5.49
1	0.115	0.0203	0.0573	0.0273	5.73
2	0.126	0.0198	0.063	0.0296	6.3
3	0.134	4.88e-10	0.069	0.0352	6.9
4	0.143	0	0.073	0.0458	7.3
5	0.137	3.93e-09	0.0762	0.043	7.62
6	0.135	4.03e-09	0.0782	0.0371	7.82
7	0.14	0	0.0786	0.0409	7.86

Table 4.9: All Five Agents Data (Large Population).

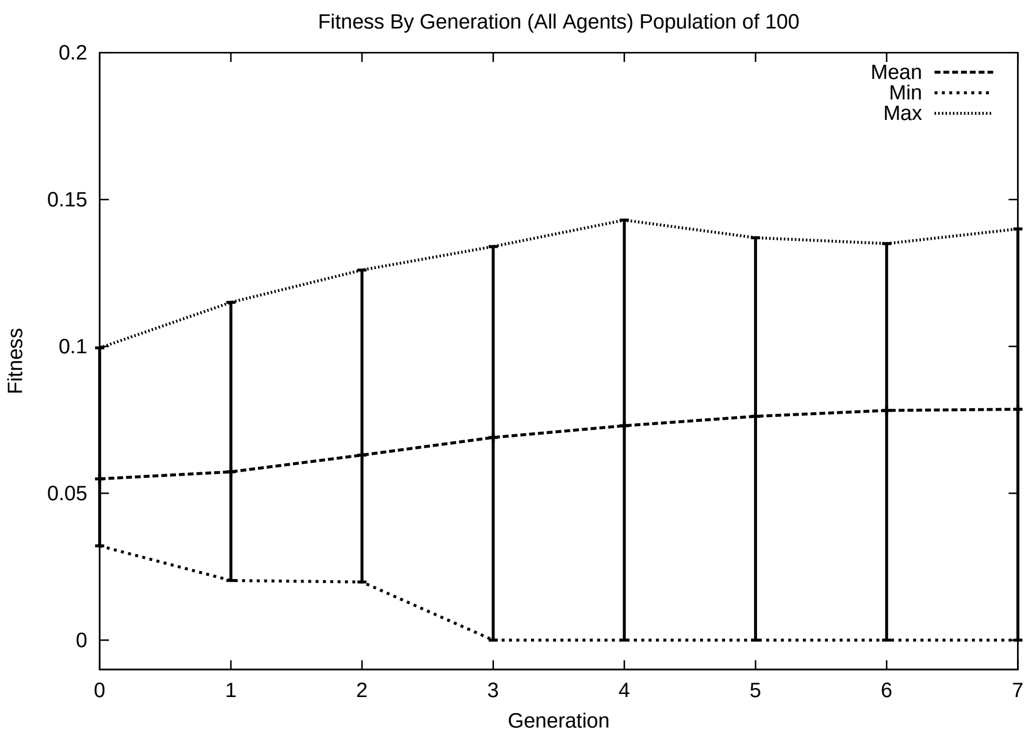


Figure 4.9: GA Data Plot With All Agents (Large Population).

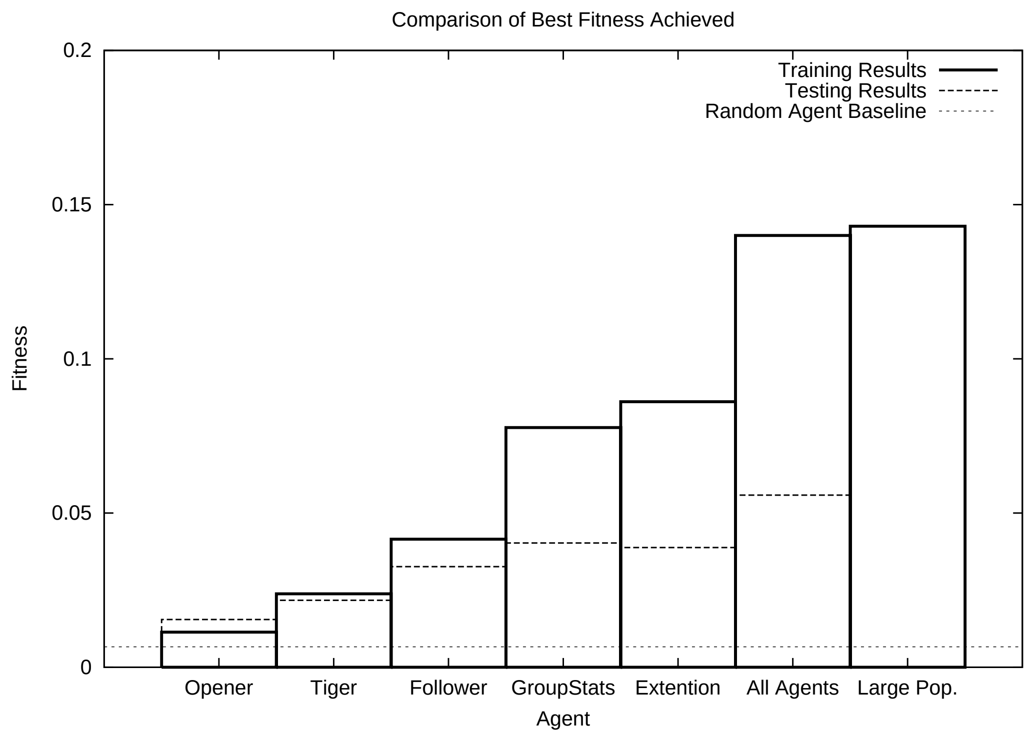


Figure 4.10: Agent Comparison.

4.3 Summary

Figure 4.10 shows a comparison of the best fitnesses achieved by all of the agent configurations. The randomly playing agent played the poorest, and the two configurations that could benefit from the genetic algorithm (extension agent and all of the agents combined) actually did . The testing data shows some variability, and in some cases an agent that performed better on the training data did worse on the testing data (compared to the other agents). Mostly, though, there appears to be a benefit of using the genetic algorithm to evolve go players.

Chapter 5

Conclusion

5.1 Contributions

We have found that a multiagent approach using a summation network does indeed yield a viable go player. Furthermore, improvement was gained over the course of multiple generations. In addition to these results, a unique approach to playing go was illustrated. As far as we know, nobody has written a program that plays go using probabilistic methods incorporating multiple agents whose interactions (the summation network) have been evolved or learned in some way. This approach shows that it may be possible to break down certain large intractable problems and use genetic algorithms to combine multiple sources of information *without knowing exactly how the information interacts to form a solution.* This architecture exemplifies the possibility of trading the ability to fine-tune the behavior of a system with the ability to scale the system indefinitely, limited mainly by the number of processing nodes.

5.2 Limitations

This approach to playing go has many potential limitations. Foremost, it relies heavily on the ability of the programmer to create agents that contribute to the skill of the program. As we are not go experts, creating good agents was a challenge.

Another limitation is that genetic algorithms take long periods of time to run. Larger training sets, larger testing sets, larger populations, more intricate summation networks, and more generations could all help improve the program, but unfortunately all of these would contribute to a significantly slower program.

Though scalability was an important goal, the realization of a massively parallel multiagent go program must be quelled by the prohibitive cost and the scarcity of machines with dozens of processors. The future may not hold a limitation such as this, but currently it is a very real limitation to increasing the number of agents extensively.

Yet another restrictive aspect of this work was the use of only 9×9 boards for all experiments. This enabled us to complete the research in a reasonable amount of time. Trying to use 19×19 boards would have likely taken too long.

The program does not play go very well, though the GA does allow the program to improve which was one of the goals of this project. Many authors often compare their programs to standard programs such as one called *Wally*, but our program does not yet have an interface that would allow automatic matches. Though this is a limitation, not playing well may not matter as much as showing that our program improves. Clearly, the program that we developed does not play go better than its peers.

5.3 Future Work

The future of this approach remains unclear, but additional research to test larger networks utilizing a larger number of agents could yield positive results. Scalability was a secondary goal—a goal that seems within reach given the prominence and proliferation of multiprocessor machines. Perhaps in the decades to come someone will create a go program that can play at the level of the masters. This is a goal that many await patiently.

Bibliography

[1] Anonymous. `http://www.daimi.au.dk/~tusk/nngs/1995-2000-go9.zip`, June 2001.

[2] S.F. da Silva. *Go and Genetic Programming: Playing Go with Filter Functions.* Master's thesis, Universiteit Leiden, Computer Science Department, November 1996.

[3] Paul Donnelly, Patrick Corr, and Danny Crookes. *Evolving Go Playing Strategy in Neural Networks.* `http://hyperion.advanced.org/18242/data/resources/nn_go.pdf`.

[4] Markus Enzenberger. *The Integration of A Priori Knowledge into a Go Playing Neural Network.* `http://www.cgl.ucsf.edu/go/Programs/neurogo-html/neurogo.html`, September 1996.

[5] David Fotland. *Knowledge Representation in the Many Faces of Go.* `http://www.smart-games.com/knowpap.txt`, February 1993.

[6] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[7] Jeffrey Greenberg. *Breeding Software to Play the Game of Go.* `http://www.inventivity.com/OpenGo/Papers/jeffg/breed.html`.

[8] Michael N. Huhns and Larry M. Stephens. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter 2, pages 79–118. The MIT Press, first edition, 1999.

[9] Takuya Kojima, Kazuhiro Ueda, and Saburo Nagano. Evolutionary algorithm extended by ecological analogy and its application to the game of go. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence(IJCAI-97)*, pages 684–689. The University of Tokyo, College of Arts and Sciences, 1997.

[10] Martin Müller. Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory. PhD dissertation, ETH Zürich, 1995.

[11] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, first edition, 1999.

[12] Bradford Nichols, Dick Buttlar, and Jackie Farrell, editors. *Pthreads Progamming*, chapter 1–2, pages 1–60. O'Reilly & Associates, Inc., first edition, 1998.

[13] Peter Norvig. *Paradigms of Artificial intelligence Programming: Case Studies in Common Lisp*, chapter 18, pages 596–653. Morgan Kaufmann Publishers, 1992.

[14] Daniel Polani and Risto Miikkulainen. Eugenic neuro-evolution for reinforcement learning. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 1041–1046, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[15] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, pages 227,616–619. The Press Syndicate of the University of Cambridge, second edition, 1997.

[16] Valluru Rao and Hayagriva Rao. *C++ Neural Networks and Fuzzy Logic*. MIS:Press, a subsidiary of Henry Holt and Company, Inc., second edition, 1995.

[17] Norman Richards, David Moriarty, and Risto Miikkulainen. Evolving neural networks to play go. *Applied Intelligence*, 8:85–96, 1998.

[18] Christopher D. Rosin and Richard K. Belew. Methods for competitive co-evolution: Finding opponents worth beating. In L.J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, La Jolla, CA, March 1995. Cognitive Computer Science Research Group, Department of Computer Science and Engineering, University of California, ICGA.

[19] Stuart J. Russell and Peter Norvig, editors. *Artificial Intelligence: A Modern Approach*, chapter 5, pages 122–145. Prentice Hall, first edition, 1995.

[20] Michael Wooldridge. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter 1, pages 28–31. The MIT Press, first edition, 1999.

Appendix A: Doxygen Code Reference

The code index was generated automatically using a tool called *Doxygen* that parses the source files' comments...

A.1 Cross-references

A.1.1 Exodus Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

A.1.2 Exodus Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

A.1.3 Exodus File List

Here is a list of all documented files with brief descriptions:

A.1.4 Exodus Related Pages

Here is a list of all related documentation pages:

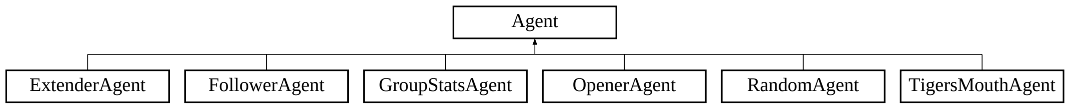
A.2 Exodus Class Documentation

A.2.1 Agent Class Reference

Defines the basic structure of an agent.

`#include <agent.h>`

Inheritance diagram for Agent::



Public Methods

- **Agent** ()

  Constructor.

- virtual ~**Agent** ()

  Destructor.

- void **set_id** (int id)

  Sets agent ID.

- int **get_id** (void)

  Gets agent ID.

- void **set_bb_ptr** (**Blackboard** *bb_p)

  Set the blackboard pointer.

- void **set_pb_ptr** (**ProbBoard** *pb_p)

  Set the probboard pointer.

- virtual void **force** (void)=0

  Force agent to make a move.

- virtual void **update** (**Game** *)=0

  Updates the game for the agent. Refresh agent with a new game state.

- virtual bool **dowork** (void)=0

  Main agent work function.

- virtual void **notify** (void *)=0

  Tell the agent something.

- virtual unsigned int **query_bits_needed_from_GA** (void)=0

  Ask the agent how many bits it needs from GA.

- virtual void **send_bits** (**chromosome_t** chrom, int start)=0

  Allows the agent to get the bits it needs.

Protected Attributes

- int **ID**

  Unique agent identification number.

- **Blackboard∗ bb_ptr**

  Blackboard (p. 77).

- **ProbBoard∗ pb_ptr**

  Probability board.

- **Game theGame**

  The game in question.

A.2.1.1 Detailed Description

Defines the basic structure of an agent.

Warning:
This is an abstract class.

A.2.1.2 Member Function Documentation

void Agent::set_id (int *id*) Sets agent ID.

Sets the agent ID number which can be used to uniquely order all agents to prevent dealocks due to possible future agent dependences and the use of thread pools. Thread pools only allow a finite number of agents to run at a time, and if the first agents to run depend on another agent that isn't running, then deadlock will occur. Agent IDs prevent this.

Warning:

IDs are currently not used, but in the event that they do become used, then it shall be expected that agents with higher IDs have dependences on only agents with lower IDs (if any at all).

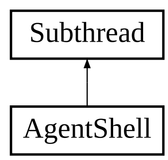
```
0074 { this->ID = id; };
```

A.2.2 AgentShell Class Reference

Represents a single thread in a thread pool.

`#include <agent.h>`

Inheritance diagram for AgentShell::



Public Methods

- **AgentShell** ()

  Constructor.

- ~**AgentShell** ()

  Destructor.

Private Methods

- void **processing** (void)

  Represents a single thread of the thread pool's main processing loop.

- void **init** (void)

  Performs any initialization that is needed.

Private Attributes

- **Agent* theAgent**

  Agent (p. 73) *identity to assume.*

A.2.2.1 Detailed Description

Represents a single thread in a thread pool.

This class shall be able to "turn" into any of the agents via the proper messages.

A.2.2.2 Constructor & Destructor Documentation

AgentShell::AgentShell () Constructor.

Note:

Stores a copy of the game, not a pointer.

```
                        {

   theAgent = NULL;
}
```

A.2.2.3 Member Function Documentation

void AgentShell::init (void) `[private]` Performs any initialization that is needed.

Note:

Currently, this function is a stub.

```
{ }
```

A.2.3 Blackboard Class Reference

This class contains globaly relavent information.

`#include <blackboard.h>`

Public Methods

- void **set_game_ptr** (**Game** *gamePtr)

  Tell the blackboard what game to look at.

- void **update** (void)

  Instructs Blackboard to update internal data-structures.

Static Private Attributes

- **Game* g_ptr**

  Points to the current game.

A.2.3.1 Detailed Description

This class contains globaly relavent information.

A blackboard is a paradigm whereby agents write information or data to a single localized location. This class provides an interface to this global scratchpad.

Warning:
Set gptr before doing anything. After this is set, all function calls are undefined until update is called at least once. Note that this class is a stub and currently provides no actual functionality.

A.2.3.2 Member Function Documentation

void Blackboard::update (void) Instructs Blackboard to update internal data-structures.

This function causes the blackboard class to regenerate all data-structures it stores locally.

```
0041 {
0042 }
```

A.2.4 Board Class Reference

Defines a goban abstraction.

`#include <board.h>`

Public Types

- enum **flags_t** { **FUNKNOWN**, **FSAFE**, **FEMPTY** }

  Flags for board capture state-machine.

Public Methods

- **Board** ()

  Constructor I.

- **Board** (const Board &other)

  Copy Constructor.

- ~**Board** ()

  Destructor.

- bool **valid_location** (**loc_t**) const

  Tells if the location is playable.

- **loc_t get_bsize** (void) const

  Gets board size.

- **Stone*** **get_goban** (void) const

  Gets goban array.

- string **raw_output** (void)

  Output function for GUI.

- pair<**usi_t**, **usi_t**> **play_move** (**loc_t**, **color_t**)

  Plays a move on the board.

- void **invert** (void)

  Inverts the stones' colors.

- bool **operator==** (Board &)

  Equality operator.

- bool **operator!=** (const Board &)

Inequality operator.

- Board **operator=** (Board)

  Assignment operator.

- **color_t get_color_played** (void)

  Returns the color played for this board.

- **loc_t get_move_played** (void)

  Gets the move that was played for this board.

- **color_t operator**[ ] (**loc_t** location)

  Offset operator.

Static Public Attributes

- const **usi_t PASS** = 0xFFFF

  Offset into board array of PASS is used to represent a pass.

- **usi_t BSIZE**

  Default board size.

- **usi_t HANDICAP**

  Size of handicap.

- list<**usi_t**> **HANDICAP_PLACES**

  Force handicap locations.

Private Methods

- void **del_stone** (**loc_t**)

  Removes a stone from board.

- **usi_t del_group** (**color_t**)

  Removes groups with no liberties.

- void **setup** ()

  Bulk of the constructors' logic.

- void **fill_safety** (vector< **flags_t** > &, int, **color_t**)

  Recursive flood for finding safe stones.

- void **put_stone** (**loc_t**, **color_t**)

  Sets a board location to a specific color.

Private Attributes

- **loc_t loc_played**

  Location played to make this board.

- **Stone* goban**

  Actual board.

- **color_t color_played**

  Who's turn is it?

- **loc_t actual_size**

  size of goban vector (BSIZE^2).

Static Private Attributes

- bool **PRINTEXTRA**

  Print extra right and bottom information on board.

Friends

- ostream& **operator**<< (ostream &strm, Board &aBoard)

  Output operator.

A.2.4.1 Detailed Description

Defines a goban abstraction.

This class stores a board as an array of **Stone** (p. 160) classes. It provides all functions that would be expected from a board abstraction.

A.2.4.2 Constructor & Destructor Documentation

Board::~Board () Destructor.

Deallocates array of **Stone** (p. 160) classes

```
            {
   //cerr << "pre " << flush;
   delete goban;
   //cerr << "post" << endl;
}
```

A.2.4.3 Member Function Documentation

usi_t Board::del_group (color_t *color*) `[private]` Removes groups with no liberties.

Parameters:

color The color of the groups to remove

```
{
   TAU_PROFILE("Board::del_group()", "", TAU_DEFAULT);

   color_t enemy_color = INV(color);
   vector<flags_t> scratch(actual_size, FUNKNOWN);

   // Mark enemy stones as safe
   for (int x=0; x<actual_size; ++x) {
      if (enemy_color==goban[x].getcolor()) scratch[x]=FSAFE;
   }


   // Do a flood fill on each empty spot, filling over friendly
   // stones but not passing enemy stones.
   for (int x=0; x<actual_size; ++x) {
      if (EMPTY==goban[x].getcolor()) {
         fill_safety(scratch, x, color);
      }
   }

```

```
   // Remove "unknown" stones as they are now dead.
   usi_t count=0;
   for (int x=0; x<actual_size; ++x) {
      if (FUNKNOWN==scratch[x]) {
         ++count;
         goban[x].setcolor(EMPTY);
      }
   }

   return count;
}
```

void Board::fill_safety (vector< flags_t > & *scratch*, int *loc*, color_t *color*) `[private]` Recursive flood for finding safe stones.

Does a flood fill of all safe pieces. Any stone that is safe automatically (logically) gives its safeness to all adjacent stones of the same color.

Parameters:

scratch A pass-by-reference scratch-pad used in this algorithm to figure what stones are safe/dead

loc Location to start at when looking for safety.

color The color to check for safety.

```
                                                                  {
    TAU_PROFILE("Board::fill_safety()", "", TAU_DEFAULT);

    if (scratch[loc] != FSAFE) {
       scratch[loc] = FSAFE;
       if (goban[loc].notleft() &&
           (color==goban[loc-1].getcolor()))
          fill_safety(scratch, loc-1, color);
       if (goban[loc].notright() &&
           (color==goban[loc+1].getcolor()))
          fill_safety(scratch, loc+1, color);
       if (goban[loc].nottop() &&
           (color==goban[loc-BSIZE].getcolor()))
          fill_safety(scratch, loc-BSIZE, color);
       if (goban[loc].notbottom() &&
           (color==goban[loc+BSIZE].getcolor()))
          fill_safety(scratch, loc+BSIZE, color);
    }
}
```

loc_t Board::get_move_played (void) Gets the move that was played for this board.

Precondition:

play_move was called already for this board.

```
{ return loc_played; }
```

void Board::invert (void) Inverts the stones' colors.

This function makes all white stones black and all black stones white.

```
                        {
  TAU_PROFILE("Board::invert()", "", TAU_DEFAULT);
  for (int x=0; x<actual_size; ++x) {
     if (goban[x].getcolor() != EMPTY) {
        goban[x].setcolor(INV(goban[x].getcolor()));
     }
  }
}
```

pair< usi_t, usi_t > Board::play_move<usi_t, usi_t> (loc_t *offset*, color_t *color*) Plays a move on the board.

Parameters:

offset Move to play

color Color to play

Returns:

A pair such that the second element is a count of the stones removed for called color and the first element is a count of the stones removed for the opposite of the called color. The first element is thus the most important.

Precondition:

color is BLACK or WHITE but not EMPTY

```
{
   TAU_PROFILE("Board::play_move()", "", TAU_DEFAULT);

   assert(offset < actual_size);

   loc_played = offset;

```

```
   goban[offset].setcolor(color);

   // Check and delete for dead of opposite color
   usi_t them = del_group(INV(color));

   // Check and delete for dead of our color
   usi_t us = del_group(color);

   // Record with this board the color of the move just played
   color_played = color;

   return make_pair(them, us);
}
```

void Board::put_stone (loc_t *loc*, color_t *color*) `[private]` Sets a board location to a specific color.

Parameters:

loc location as a single-dimention array offset

color The color of the stone to place

Warning:

Does not check for captures or suicide

```
                                              {
    goban[loc].setcolor(color);
}
```

string Board::raw_output (void) Output function for GUI.

Raw board output

```
{
   string tmp;

   //parsable board
   tmp += "board ";
   for (int loc=0; loc<actual_size; ++loc) {
       switch (goban[loc].getcolor()) {
          case BLACK: tmp += "B"; break;
          case WHITE: tmp += "W"; break;
          case EMPTY: tmp += "N"; break;
       }
       if ( (loc != (actual_size-1)) &&
            (( BSIZE - 1) == (loc % BSIZE ))) tmp += ":";
```

```
    }
    tmp += "\n";

    return tmp;
}
```

void Board::setup () `[private]` Bulk of the constructors' logic.

This function performs the actual setup of the board. It allocates the **Stone** (p. 160) class array and sets variables to initial values.

Precondition:

size is a natural number, and all elements in the list handicapPlaces are less than size*size. A board smaller than three or four probably is not useful as well.

Postcondition:

All variables are initialized and goban especially is setup. The exception is the variable loc_played which is undefined.

Warning:

loc_played is defined upon exit as the last of the setup moves played. If the board starts with a move at A13 then H2 as a handicap, then H2 is the logical value stored here.

```
{

    // Allocate board and define its size
    actual_size = BSIZE * BSIZE;
    //bsize=size;

    goban = new Stone [actual_size];

     // Check memory allocation
    if (!goban) {
       LOG("-BRD       -E- Goban memory allocation failed.");
       cerr << "-E- Goban memory allocation failed.";
       exit(1);
    }

    PRINTEXTRA = false;

    // Setup which turn.  If a non-handicap game, black plays first on this
    // board which means that the "next turn" is white.  On the other hand,
    // if there is a handicap, white plays first as black's handicap was his
    // virtual first move.
```

```
   if (HANDICAP_PLACES.empty()) {
      color_played=WHITE;
   } else {
      color_played=BLACK;
   }

   // Setup each spot in goban as empty (also setup column/row information)
   Stone tmp_stone;
   int col, row;
   for (int x=0; x < actual_size; ++x) {
      tmp_stone.clear();

      // Offset mod the board size yield column number
      col=x%BSIZE;

      // Offset divided by board size yields row number when truncated
      row=static_cast<usi_t>(x / BSIZE);

      tmp_stone.setcol(col);
      tmp_stone.setrow(row);
      if (col == (BSIZE - 1)) tmp_stone.setlastcol();
      if (row == (BSIZE - 1)) tmp_stone.setlastrow();

      goban[x] = tmp_stone;
   }

   // Setup handicaps
   std::list<loc_t>::iterator pos;
   for (pos=HANDICAP_PLACES.begin(); pos != HANDICAP_PLACES.end(); ++pos) {
      // Only black gets handicap stones
      goban[*pos].setcolor(BLACK);
      loc_played = *pos;
   }
}
```

bool Board::valid_location (loc_t *loc*) const Tells if the location is playable.

This function takes no rules into account other than "cannot play on an already taken spot."

Parameters:

loc offset into board vector

Warning:

Doesn't check for loc less than zero, but it's unsigned so it doesn't matter.

```
{
   TAU_PROFILE("Board::valid_location()", "", TAU_DEFAULT);
```

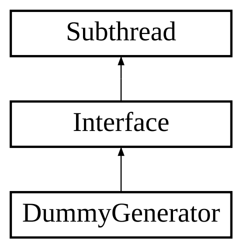
```

   return ((loc < actual_size) && goban[loc].empty());
}
```

A.2.5 DummyGenerator Class Reference

A dummy move generator that generates random legal moves.

```
#include <outputgen.h>
```

Inheritance diagram for DummyGenerator::



Public Methods

- **DummyGenerator** ()

  Constructor.

Private Methods

- void **processing** (void)

  Main processing function.

Private Attributes

- unsigned int **rndbuf**

  State variable for random number generation.

A.2.5.1 Detailed Description

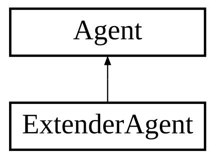
A dummy move generator that generates random legal moves.

A.2.6 ExtenderAgent Class Reference

Suggests moves that extend from friendly stones.

```
#include <agent.h>
```

Inheritance diagram for ExtenderAgent::



Public Methods

- **ExtenderAgent** ()

  Constructor.

- ~**ExtenderAgent** ()

  Destructor.

- void **force** (void)

  Force this agent to move.

- void **update** (**Game** ∗)

  Updates the game for the agent. Refresh agent with a new game state.

- bool **dowork** (void)

  work thread.

- void **notify** (void ∗)

  Tell the agent something.

- unsigned int **query_bits_needed_from_GA** (void)

  Asks the agent how many bits it needs in the GA.

- void **send_bits** (**chromosome_t** chrom, int start)

  Sends to this agent the bits it needs from the GA.

Private Methods

- void **attempt** (int value, int locations[ ], int start)
  A helper function.
- unsigned int **getval** (**chromosome_t** chrom, int start)
  Calculates extention value from chromosome.

Private Attributes

- unsigned int **bits_per_value**
  Number of GA bits to use for each of the extention types.
- unsigned int **num_values**
  Number of extention types below.
- int **extendValue**
  Simple extention value.
- int **extendLocations** [5]
  Locations.
- int **onePointExtendValue**
  1-point extention value.
- int **onePointExtendLocations** [5]
  Locations.
- int **twoPointExtendValue**
  2-point extention value.
- int **twoPointExtendLocations** [5]
  Locations.
- int **threePointExtendValue**
  3-point extention value.
- int **threePointExtendLocations** [5]

Locations.

- int **shoulderValue**

  Shoulder extention value.

- int **shoulderLocations** [5]

  Locations.

- int **knightValue**

  Knight's move value.

- int **knightLocations** [9]

  Locations.

- int **largeKnightValue**

  Large knight's move value.

- int **largeKnightLocations** [9]

  Locations.

A.2.6.1 Detailed Description

Suggests moves that extend from friendly stones.

A.2.6.2 Member Function Documentation

unsigned int ExtenderAgent::getval (chromosome_t *chrom*, int *start*)
`[private]` Calculates extention value from chromosome.

Parameters:

chrom The chromosome

start Offset in chromosme where ExtenderAgent parameters are stored.

Returns:

The value read from the chromosome as an integer

```
{
    int sum = 0;

    for(unsigned int x=start; x<start+bits_per_value; x++) {
       sum += static_cast<int>(chrom[x] * pow(2, x-start));
    }

    return sum;
}
```

unsigned int ExtenderAgent::query_bits_needed_from_GA (void) `[virtual]` Asks the agent how many bits it needs in the GA.

Returns:

Number of bits needed in GA chromosome

Reimplemented from **Agent** (p. 75).

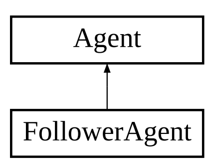
```
{
   return bits_per_value * num_values;
}
```

A.2.7 FollowerAgent Class Reference

Suggests moves near opponent's last move.

`#include <agent.h>`

Inheritance diagram for FollowerAgent::



Public Methods

- **FollowerAgent** ()

  Constructor.

- **~FollowerAgent** ()

Destructor.

- void **force** (void)

  Force the agent to make its move.

- void **update** (**Game** *)

  Update this agent with the latest state of the game.

- bool **dowork** (void)

  work thread.

- void **notify** (void *)

  Tell the agent something.

- unsigned int **query_bits_needed_from_GA** (void)

  Tells how many bits this agent needs from the GA.

- void **send_bits** (**chromosome_t** chrom, int start)

  Sends to this agent the bits it needs from the GA.

Private Methods

- void **imprint** (**loc_t** loc, **Board** &b)

  Helper function for internal algorithm.

A.2.7.1 Detailed Description

Suggests moves near opponent's last move.

A.2.7.2 Member Function Documentation

void FollowerAgent::imprint (loc_t *loc*, Board & *b*) `[private]` Helper function for internal algorithm.

Adds a probability to a location based on how close it is to enemy stones

Parameters:

loc The location to check

b The go board to consult

```
{
    Stone *stones = b.get_goban();
    Stone s = stones[loc];
    color_t color = theGame.get_turn();

    if (color == BLACK) {
      if (s.notleft()) {
         // Due left
         if (stones[loc-1].white()) (*pb_ptr)[loc] += 1.0;

         // Top left
         if (s.nottop() && stones[loc-Board::BSIZE-1].white()) {
            (*pb_ptr)[loc] += 0.5;
         }

         // Bottom left
         if (  s.notbottom() &&
               stones[loc+Board::BSIZE-1].white()) {
            (*pb_ptr)[loc] += 0.5;
         }

         // Level 2: TODO
      }
      if (s.notright()) {
         // Due right
         if (stones[loc+1].white()) (*pb_ptr)[loc] += 1.0;

         // Top right
         if (  s.nottop() &&
               stones[loc-Board::BSIZE+1].white()) {
            (*pb_ptr)[loc] += 0.5;
         }

         // Bottom right
         if (  s.notbottom() &&
               stones[loc+Board::BSIZE+1].white()) {
            (*pb_ptr)[loc] += 0.5;
         }

         // Level 2: TODO
      }
      if (s.nottop()) {
         // Due top
         if (stones[loc-Board::BSIZE].white())
                  (*pb_ptr)[loc] += 1.0;
         // level 2

      }
      if (s.notbottom()) {
```

```
       // Due bottom
       if (stones[loc+Board::BSIZE].white())
               (*pb_ptr)[loc] += 1.0;
       // level 2
    }
  } else {
    if (s.notleft()) {
       // Due left
       if (stones[loc-1].black()) (*pb_ptr)[loc] += 1.0;

       // Top left
       if (s.nottop() && stones[loc-Board::BSIZE-1].black()) {
          (*pb_ptr)[loc] += 0.5;
       }

       // Bottom left
       if (s.notbottom() &&
           stones[loc+Board::BSIZE-1].black()) {
          (*pb_ptr)[loc] += 0.5;
       }

       // Level 2: TODO
    }
    if (s.notright()) {
       // Due right
       if (stones[loc+1].black()) (*pb_ptr)[loc] += 1.0;

       // Top right
       if (s.nottop() && stones[loc-Board::BSIZE+1].black()) {
          (*pb_ptr)[loc] += 0.5;
       }

       // Bottom right
       if (s.notbottom() && stones[loc+Board::BSIZE+1].black()) {
          (*pb_ptr)[loc] += 0.5;
       }

       // Level 2: TODO
    }
    if (s.nottop()) {
       // Due top
       if (stones[loc-Board::BSIZE].black()) (*pb_ptr)[loc] += 1.0;
       // level 2

    }
    if (s.notbottom()) {
       // Due bottom
       if (stones[loc+Board::BSIZE].black()) (*pb_ptr)[loc] += 1.0;
       // level 2
    }
  }
```

```
}
```

A.2.8 Ga Class Reference

Defines a Genetic Algorithm.

```
#include <ga.h>
```

Public Methods

- **Ga** ()

  Constructor.

- ~**Ga** ()

  Destructor.

- void **init** ()

  Initialize first generation.

- void **set_codex** (**PreCodex** *)

  Tells GA what class has the fitness function to use.

- int **loadpop** (string name=””)

  Loads a generation from disk.

- int **savepop** (string name=””)

  Saves a generation to disk.

- int **savebest** (string name=””)

  Saves the best chromosome to disk.

- void **start** (void)

  Starts the GA process.

Static Public Attributes

- **usi_t MAXGEN**

  Maximum number of generations.

- **usi_t POPSIZE**

  Size of the population.

- float **FITNESS_CUTOFF**

  Unused.

- float **PCROSS**

  Probability of crossover.

- float **PMUTATION**

  Probability of mutation.

- float **FMULTIPLE**

  Linear scaling parameter.

- string **FILENAME_IN**

  For loading generations from disk.

- string **FILENAME_OUT**

  For saving generations to disk.

- string **BEST_FILENAME_OUT**

  For saving a chromosome.

- string **TRAIN_FILE**

  SGF-derived data for training.

Private Methods

- void **ftest** (float *f, float *prob)

  Compute the F-test.

- void **ttest** (float *t, float *prob)

  Compute (Student's) T-test.

- void **tutest** (float *t, float *prob)

Compute the T-test (Student's) if the variances aren't the same.

- int **select** (**Population** &pop)

  selects a chromosome.

- bool **flip** (float)

  Bernoulli probability.

- **allele mutation** (**allele** alleleval)

  Mutates an allele.

- void **crossover** (**chromosome_t** &parent1, **chromosome_t** &parent2, **chromosome_t** &child1, **chromosome_t** &child2, **usi_t** &jcross)

  Crosses the two parents to create the children.

- void **generation** (void)

  Increments the generation.

- float **scale** (float, float, float)

  Scales the fitness.

- void **scalepop** (void)

  Scales the fitness of the population.

- void **prescale** (float &, float &)

  Calculates linear parameters.

Private Attributes

- pthread_mutex_t **interrupt_watcher**
- **Population∗ oldpop**

  Pointer to old population.

- **Population∗ newpop**

  Pointer to new population.

- **usi_t lchrom**

Length of a chromosome.

- **usi_t gen**

  Generation counted.

- bool **stop**

  Interrupt flag.

- **usi_t nmutation**

  Number of mutations performed.

- **usi_t ncross**

  Number of crossovers performed.

- float **osumfitness**

  1st generation fitness sum.

- float **oavg**

  Average fitness in the first generation.

- float **omax**

  Maximum fitness in the first generation.

- float **omin**

  Minimum fitness in the first generation.

- float **ostdev**

  Standard deviation in the first generation.

- float **ovar**

  Variance in the first generation.

- unsigned int **rndbuf**

  Used for thread-safe random number generation.

- **PreCodex* leader**

  Points to the class that has the fitness function get_fitness().

A.2.8.1 Detailed Description

Defines a Genetic Algorithm.

The genetic algorithm needs a fitness function, thus one must make a call to **set_codex**() (p. 108) before this class can be used.

A.2.8.2 Member Function Documentation

bool Ga::flip (float *val*) `[private]` Bernoulli probability.

Returns:

True if val is greater or equal to a uniform pseudo-random variable generated over [0,1]

```
{
  TAU_PROFILE("Ga::flip()", "", TAU_DEFAULT);
  float res = static_cast<float>(rand_r(&rndbuf)) /
              static_cast<float>(RAND_MAX) ;
  return (res <= val) ? true : false;
}
```

void Ga::ftest (float * *f*, float * *prob*) `[private]` Compute the F-test.

Computes statistics that help evaluate whether two distributions have different variances.

Author:

Numerical Recipes in C, modified by Todd Blackman, page 619

```
{

  if (ovar > newpop->var) {
    *f = ovar / newpop->var;
    if ( fabs(newpop->var - 0.0) < 0.0000001) {
       if (!global_data.reg_on) {
         cerr << "-E- Bad variance of zero." << endl;
         LOG("-E- Bad variance of zero.");
       }
       *f = -1;
       *prob = -1;
       return;
    }
  } else {
    *f = newpop->var / ovar;
```

```
     if ( fabs(ovar - 0.0) < 0.0000001) {
        if (!global_data.reg_on) {
          cerr << "-E- Bad variance of zero." << endl;
          LOG("-E- Bad variance of zero.");
        }
        *f = -1;
        *prob = -1;
        return;
     }
   }
   float df = POPSIZE - 1;

   *prob = 2.0 * betai(0.5*df, 0.5*df, df/(df + df * (*f)));
   if (*prob > 1.0) {
      *prob = 2.0 - *prob;
   }

}
```

void Ga::init (void) Initialize first generation.

This function creates the newpop and oldpop structures which are both identical after this function finishes.

```
              {
   assert(leader != 0);

   //LOG("- GA       -M- Entered init().");

   oldpop->sumfitness = 0.0;
   oldpop->stdev = 0.0;
   oldpop->var = 0.0;
   oldpop->max = 0.0;
   oldpop->min = 10000;

   gen = 0;

   // Loop through each individual and initialize it
   for (int i=0; i<POPSIZE; ++i) {
      Individual indv;

      assert(indv.chrom.empty());

      for (int a=0; a<lchrom; ++a) {
         indv.chrom.push_back(flip(0.5));
      }
      indv.parent1=0;
      indv.parent2=0;
      indv.xsite=0;

```

```
      indv.fitness=leader->get_fitness(indv.chrom);
      indv.ofitness = indv.fitness;

      assert(indv.fitness >= 0.0);

      oldpop->individuals.push_back(indv);

      oldpop->sumfitness += indv.fitness;

      if ((*oldpop).individuals[i].fitness > oldpop->max) {
         oldpop->max = (*oldpop).individuals[i].fitness;
         oldpop->whichmax = i;
      }
      if ((*oldpop).individuals[i].fitness < oldpop->min) {
         oldpop->min = (*oldpop).individuals[i].fitness;
         oldpop->whichmin = i;
      }

      // This is to make the other vector know its size
      newpop->individuals.push_back(indv);
   }

   //assert(oldpop->sumfitness > 0.0);
   //assert(oldpop->max > 0.0);

   oldpop->avg = oldpop->sumfitness / POPSIZE;

   // Calculate the standard deviation of the values
   float diffsquare;
   for (int i=0; i<POPSIZE; ++i) {
      diffsquare = (*oldpop).individuals[i].fitness -
                   oldpop->avg;
      diffsquare *= diffsquare;
      oldpop->var += diffsquare;
   }
   oldpop->var /= POPSIZE - 1;
   oldpop->stdev = sqrt(oldpop->var);

   newpop->avg = oldpop->avg;
   newpop->max = oldpop->max;
   newpop->min = oldpop->min;
   newpop->sumfitness = oldpop->sumfitness;
   newpop->stdev = oldpop->stdev;
   newpop->var = oldpop->var;
   newpop->whichmax = oldpop->whichmax;
   newpop->whichmin = oldpop->whichmin;

   osumfitness = newpop->sumfitness;
   oavg = newpop->avg;
   omax = newpop->max;
   omin = newpop->min;
```

```
    ostdev = newpop->stdev;
    ovar = newpop->var;
}
```

int Ga::loadpop (string *name* = "") Loads a generation from disk.

Parameters:

name The file name containing the population

Author:

Todd Blackman

```
{
    int chrom_length;
    int pop_size;
    chromosome_t tmpchrome;
    Individual tmpindev;
    char c;
    //int x=0;

    if (leader == NULL) {
       cout << "-E- use set_codex() first." << endl;
       exit(1);
    }

    if (name == "") name = Ga::FILENAME_IN;

    oldpop->individuals.clear();
    newpop->individuals.clear();

    newpop->max = 0.0;
    newpop->min = 10000;
    newpop->sumfitness = 0.0;

    //LOG("- GA       -M- Loading.");
    ifstream fin(name.c_str());
    if (fin) {
       fin >> pop_size;
       fin >> chrom_length;

       if (pop_size != POPSIZE) {
         cerr << "-E- datafile population size doesn't match." << endl;
          cerr << POPSIZE << endl;
          cerr << pop_size << endl;
          LOG("- GA       -E- Invalid popsize or chromosome_t length read");
          return -1;
       }
       if (chrom_length != lchrom) {
```

```
      cerr << "-E- datafile chromosome length doesn't match." << endl;
      cerr << lchrom << endl;
      cerr << chrom_length << endl;
      LOG("- GA       -E- Invalid popsize or chromosome_t length read");
      return -1;
   }

   int i=0;
   fin.get(c); // newline grab
   while (!fin.eof()) {
      fin.get(c);
      if (c == '1') tmpchrome.push_back(true);
      if (c == '0') tmpchrome.push_back(false);
      if (c == '\n') {
         i++;
         tmpindev.chrom = tmpchrome;
         //LOG("- GA       -M- chrome is " << tmpchrome);
         tmpindev.parent1 = 0;
         tmpindev.parent2 = 0;
         tmpindev.xsite = 0;

         // Fitness and stats calculate
         tmpindev.fitness =
                 leader->get_fitness(tmpindev.chrom);
         tmpindev.ofitness = tmpindev.fitness;
         assert(tmpindev.fitness >= 0.0);
         newpop->sumfitness += tmpindev.fitness;
         if (tmpindev.fitness > newpop->max) {
            newpop->max = tmpindev.fitness;
            newpop->whichmax = i;
         }
         if (tmpindev.fitness < newpop->min) {
            newpop->min = tmpindev.fitness;
            newpop->whichmin = i;
         }


         //oldpop->individuals[x++] = tmpindev;
         if (tmpchrome.size() > 0) {
           newpop->individuals.push_back(tmpindev);
           oldpop->individuals.push_back(tmpindev);
         }
         //LOG("- GA       -M- iter10.");
         tmpchrome.clear();
      }
   }
   fin.close();

   newpop->avg = newpop->sumfitness / POPSIZE;

   scalepop();
```

```

      // Calculate the standard deviation of the values
      float diffsquare;
      newpop->stdev = 0.0;
      newpop->var = 0.0;
      for (int i=0; i<POPSIZE; ++i) {
         diffsquare = (*newpop).individuals[i].fitness -
                      newpop->avg;
         diffsquare *= diffsquare;
         newpop->var += diffsquare;
      }
      newpop->var /= POPSIZE - 1;
      newpop->stdev = sqrt(newpop->var);

      osumfitness = newpop->sumfitness;
      oavg = newpop->avg;
      omax = newpop->max;
      omin = newpop->min;
      ostdev = newpop->stdev;
      ovar = newpop->var;

      //LOG("- GA       -M- Checking loaded values "
      //<< "for integrety.");

      return 0;

   } else {
      LOG("- GA       -E- Error opening datafile " << name);
      return -1;
   }

}
```

void Ga::prescale (float & *a*, float & *b*) `[private]` Calculates linear parameters.

Parameters:

a Slope

b Intercept

Note:

Page 79

```
{
   TAU_PROFILE("Ga::prescale()", "", TAU_DEFAULT);
   float delta;

```

```
    //LOG("- GA       -M- Entering prescale().");

    // Non-negative test
    if (newpop->min > ((FMULTIPLE * (newpop->avg) -
                        (newpop->max)) /
                        (FMULTIPLE - 1.0))) {
       delta = (newpop->max) - (newpop->avg);
       if (delta == 0) {
          a = 1;
          b = 0;
       } else {
         a = (FMULTIPLE - 1.0) * (newpop->avg) / delta;
         b = (newpop->avg) *
             ((newpop->max) - FMULTIPLE * (newpop->avg)) /
             delta;
       }

    // Negative. Scale as much as possible
    } else {
       delta = (newpop->avg) - (newpop->min);
       if (delta == 0) {
          a = 1;
          b = 0;
       } else {
         a = (newpop->avg) / delta;
         b = -(newpop->min) * (newpop->avg) / delta;
       }
    }

    //LOG("- GA       -M- Prescale a = " << a << "    b = "
    //<< b << "   delta = " << delta);
}
```

int Ga::savebest (string *name* = "") Saves the best chromosome to disk.

Warning:
Untested

```
{
    if (name == "") name = Ga::BEST_FILENAME_OUT;

    ofstream fout(name.c_str());
    if (fout) {
       fout << lchrom << endl;
       fout << newpop->individuals[newpop->whichmax].chrom << endl;
       fout.close();
       return 0;
    } else {
```

```
        LOG("- GA       -E- Error opening best chromosome output datafile "
            << name);
        return -1;
    }

}
```

float Ga::scale (float *obj*, float *a*, float *b*) `[private]` Scales the fitness.

Note that the code in the text does not cut off at zero

Parameters:

obj objective value to scale

Note:

See page 79 of "Genetic Algorithms in Search, Optimization, and Machine Learning"

```
{
    TAU_PROFILE("Ga::scale()", "", TAU_DEFAULT);

    //LOG("- GA       -M- Entering scale().");

    float res;
    res = a * obj + b;
    if (res < 0.0) {
        return 0.0;
    } else {
        //LOG("- GA       -M- a = " << a << "  obj = "
        // << obj << "    b = " << b);
        assert(res >= 0.0);
        return res;
    }
}
```

int Ga::select (Population & *pop*) `[private]` selects a chromosome.

This function selects a chromosome based on a roulette wheel paradigm

Note:

Taken from page 63.

```
{
    TAU_PROFILE("Ga::select()", "", TAU_DEFAULT);

```

```
    float randn;         //<! Point on roulette wheel
    float partsum = 0.0; //<! Accumulator
    int j=0;             //<! LCV (population index)

    //LOG("- GA       -M- Entered select()");

    // Wheel location
    randn = static_cast<float>(rand_r(&rndbuf)) /
            static_cast<float>(RAND_MAX) * pop.sumfitness;

    //LOG("- GA       -M- randn = " << randn);
    //LOG("- GA       -M- sumfitness = " << pop.sumfitness);
    //LOG("- GA       -M- rndbuf = " << rndbuf);
    //LOG("- GA       -M- RAND_MAX = " << RAND_MAX);

    assert(randn <= pop.sumfitness);
    assert(pop.sumfitness >= 0);

    // Find which individual it landed on.
    do {
       partsum += pop.individuals[j++].fitness;
    //} while ((partsum < randn) && (j != POPSIZE));
    } while ((partsum < randn) && (j < POPSIZE));

    //LOG("- GA       -M- Leaving select() with value of " << j-1);

    assert(j-1 >= 0);
    assert(j-1 < POPSIZE);

    // Return the index of the individual
    return (j-1);
}
```

void Ga::set_codex (PreCodex ∗ *f*) Tells GA what class has the fitness function to use.

This function also obtains the length of the chromosome.

Parameters:

f A pointer to a class of type **PreCodex** (p. 154) through inheritance.

Author:

Todd Blackman

```
{
    leader = f;

    lchrom = leader->get_chrom_size();
```

```
   assert(lchrom < 10000);
}
```

void Ga::start (void) Starts the GA process.

Author:
Todd Blackman

```
{
   pthread_mutex_lock(&interrupt_watcher);
   stop=false;
   pthread_mutex_unlock(&interrupt_watcher);

   LOG("- GA       -M- Using population size of " << POPSIZE);
   LOG("- GA       -M- Using maximum generation of " << MAXGEN);
   LOG("- GA       -M- Using crossover percentage of " << PCROSS);
   LOG("- GA       -M- Using mutation percentage of " << PMUTATION);
   LOG("- GA       -M- Using F multiplier of " << FMULTIPLE);
   LOG("- GA       -M- Chromosome length is " << lchrom);

   // could put a mutex in loop, but so what if we read the wrong value.  On
   // the next loop iteration it will read the correct one.
   LOG("- GA       -M-  gen         max         min        mean       stdev\
  sumfitness     F-value      F-prob     T-value      T-prob");
   LOG("- GA       -M- ----  ----------  ----------  ----------  ----------\
  ----------  ----------  ----------  ----------  ----------");

   // Compute statistics
   float t,f;
   float tprob, fprob;
   ftest(&f, &fprob);
   string tsig, fsig;

   if (fprob < SIGCUTOFF) {
     tutest(&t, &tprob);
     fsig = "diff";
   } else {
     ttest(&t, &tprob);
     fsig = "same";
   }

   if (tprob < SIGCUTOFF) {
     tsig = "diff";
   } else {
     tsig = "same";
   }

```

```
// Output statstics
LOG("- GA       -M- " << setprecision(3)
     << setw(4) << gen << "  "
     << setw(10) << (newpop->max) << "  "
     << setw(10) << (newpop->min) << "  "
     << setw(10) << (newpop->avg) << "  "
     << setw(10) << (newpop->stdev) << "  "
     << setw(10) << (newpop->sumfitness) << "  "
     << "      " << fsig << "  "
     << setw(10) << fprob << "  "
     << "      " << t << "  "
     << setw(10) << tprob << "  "
     );

do {
  generation();

  // Compute statistics
  ftest(&f, &fprob);

  if (fprob < SIGCUTOFF) {
    tutest(&t, &tprob);
    fsig = "diff";
  } else {
    ttest(&t, &tprob);
    fsig = "same";
  }
  if (tprob < SIGCUTOFF) {
    tsig = "diff";
  } else {
    tsig = "same";
  }

  // Output statistics
LOG("- GA       -M- " << setprecision(3)
     << setw(4) << gen << "  "
     << setw(10) << (newpop->max) << "  "
     << setw(10) << (newpop->min) << "  "
     << setw(10) << (newpop->avg) << "  "
     << setw(10) << (newpop->stdev) << "  "
     << setw(10) << (newpop->sumfitness) << "  "
     << "      " << fsig << "  "
     << setw(10) << fprob << "  "
     << setw(10) << t << "  "
     << setw(10) << tprob << "  "
     );

  pthread_mutex_lock(&log_mutex);
  assert(thread_count <= MAX_THREADS);
  assert(thread_count == 1);
  pthread_mutex_unlock(&log_mutex);
```

```

    } while(!stop && gen<MAXGEN);

    leader->summary(newpop);
}
```

void Ga::ttest (float * *t*, float * *prob*) `[private]` Compute (Student's) T-test.

Computes statistics that help evaluate whether two distributions have different means

Author:

Numerical Recipes in C, modified by Todd Blackman, page 616

```
{
    float df,svar;

    df=POPSIZE+POPSIZE-2;

    // Compute pooled variance
    svar = ((POPSIZE-1)*ovar+(POPSIZE-1)*newpop->var)/df;
    if ( fabs(svar - 0.0) < 0.0000001) {
       *t = -1;
       *prob = -1;
       return;
    }
    *t = (oavg-newpop->avg)/sqrt(svar*(1.0/POPSIZE+1.0/POPSIZE));
    *prob=betai(0.5*df,0.5,df/(df+(*t)*(*t)));
}
```

void Ga::tutest (float * *t*, float * *prob*) `[private]` Compute the T-test (Student's) if the variances aren't the same.

Computes statistics that help evaluate whether two distributions have different means

Author:

Numerical Recipes in C, modified by Todd Blackman, page 617-8

```
{
    float df;

    *t = (oavg-newpop->avg) /
         sqrt(ovar/POPSIZE + newpop->var/POPSIZE);
```

```
    // Degrees of freedom calculation
    df=SQR(ovar/POPSIZE + newpop->var/POPSIZE) /
       (SQR(ovar/POPSIZE)/(POPSIZE-1) +
        SQR(newpop->var/POPSIZE)/(POPSIZE-1));

    *prob=betai(0.5*df, 0.5, df/(df+SQR(*t)));
}
```

A.2.8.3 Member Data Documentation

pthread_mutex_t Ga::interrupt_watcher [private] MUTEX for interrupting GA

Warning:

(unused)

A.2.9 Game Class Reference

A class that defines a series of boards.

#include <game.h>

Public Methods

- **Game** (void)

  Constructor I.

- **Game** (Game &other)

  Copy Constructor.

- ~**Game** ()

  Destructor.

- void **reset** (void)
- void **play_move** (**loc_t** l)

  Play a move.

- void **play_move** (int x, int y)

  Plays a move given (x,y) coordinates.

- void **play_move** (**move_t** m)

  Plays a move given a **move_t** (p. 148) *struct.*

- void **retract** (**usi_t** num)

  Retracts moves.

- **move_t last** (void)

  Returns the last move made.

- bool **legal** (**loc_t**)
- bool **legal** (int, int)

  Is the move legal?

- bool **is_over** (void)

  Is the game over yet?

- **Board get_board** () const

  Returns the current board.

- **usi_t get_bsize** () const

  Returns the board size.

- **color_t wturn** ()

  Whose turn is it?

- **color_t get_turn** ()

  Whose turn is it?

- void **set_turn** (**color_t** c)

  Overide game conventions and just set whose turn it is.

- list<**loc_t**> **enumerate_legal_locations** (void)

  Returns legal locations.

- void **invert_board** (void)

  Changes black to white and vice versa.

- void **lock** (void)

- void **unlock** (void)

    Unlocks the class.

- int **get_captures** (**color_t** col)

    Stub.

- int **movenum** (void)
- bool **operator==** (const Game &)

    Equality operator.

- bool **operator!=** (const Game &)
- Game **operator=** (Game)

    Assignment operator.

Static Public Methods

- void **set_super_ko** (bool a)

    Set super KO checking.

- void **set_suicide** (bool a)

    Set suicide checking.

Static Public Attributes

- bool **SUPER_KO**

    Is superko rule in affect?

- bool **SUICIDE**

    Is suicide allowed?

- float **KOMI**

    Komi points to give.

- **usi_t INITIAL_TIME**

    Inital game time.

- **usi_t BYOMI_TIME**

  Time per byomi period.

- **usi_t BYOMI_STONES**

  Stones per byomi period.

Private Methods

- void **inv_turn** (void)

  Change whose turn it is.

- void **setup** ()

  Initializes things.

Private Attributes

- list<**Board**> **theGame**

  Actual list of boards.

- **color_t whose_turn**

  Color whose turn it is.

- list<**Board**>::iterator **currentBoard**

  Iterator pointing to board.

- list<**usi_p**> **capStones**

  captured stones. 1st is black; 2nd white.

- bool **enum_memoize_flag**

  Used for memoizing legal moves.

- pthread_mutex_t **mutex**

  MUTEX for operating on internal structures.

Static Private Attributes

- bool **super_ko**

  Is superko rule in affect?

- bool **suicide**

  Is suicide allowed?

- float **komi**

  Points to white for having to play second.

Friends

- ostream& **operator**<< (ostream &strm, Game &aGame)

  Stream operator.

A.2.9.1 Detailed Description

A class that defines a series of boards.

This class stores the game as a linked-list of **Board** (p. 78) classes.

A.2.9.2 Member Function Documentation

list< loc_t > Game::enumerate_legal_locations<loc_t> (void) Returns legal locations.

Returns:
a list of integers (locations)

```
{
  TAU_PROFILE("Game::enumerate_legal_locations()", "", TAU_DEFAULT);

  list<loc_t> tmp;

  //int lcnt = 0;

  //LOG("-GAM        -M- Entered enumerate_legal_locations().");

  for (loc_t x=0; x<(Board::BSIZE * Board::BSIZE); ++x) {
```

```
       if (legal(x)) {
          tmp.push_back(x);
          //lcnt++;
       }
    }

    //LOG("-GAM        -M- Did enumerate_legal_locations(): There were "
    //            << lcnt << " legal moves excluding passing.");

   return tmp;
}
```

int Game::get_captures (color_t *col*) Stub.

Returns how many stones captured by "col"

Parameters:

col Color that has captured the other color's stones

```
{
   TAU_PROFILE("Game::get_captures()", "", TAU_DEFAULT);

   usi_p captures;

   captures.first = 0;
   captures.second = 0;

   for (list<usi_p>::iterator pos = capStones.begin();
    pos != capStones.end();
    pos++) {
     captures.first += pos->first;
     captures.second += pos->second;
   }

   if (col == BLACK) {
     return captures.first;
   } else if (col == WHITE) {
     return captures.second;
   } else {
     return -1;
   }
}
```

void Game::invert_board (void) Changes black to white and vice versa.

This function does not alter whose turn it it nor the number of stones captured semantics. The board is inverted using a call to the **Board** (p. 78) class invert function.

```
{
  TAU_PROFILE("Game::invert_board()", "", TAU_DEFAULT);

  //Board b = *theGame.rbegin();
  Board b = theGame.back();
  theGame.pop_back();
  b.invert();
  theGame.push_back(b);
}
```

bool Game::is_over (void) Is the game over yet?

This is determined by there being two passes (two identical boards in a row)

```
                    {
  TAU_PROFILE("Game::is_over()", "", TAU_DEFAULT);
  list<Board>::reverse_iterator pos1;
  list<Board>::reverse_iterator pos2;
  list<Board>::reverse_iterator pos3;

  pos1 = theGame.rbegin();
  pos2 = theGame.rbegin();
  pos2++;

  if ((theGame.empty()) ||
      (pos1 == theGame.rend()) ||
      (pos2 == theGame.rend())) {
    return false;
  } else {
     pos3 = pos2;
     pos3++;
     // Only two moves in record
     if (pos3 == theGame.rend()) {
       return false;
     } else {
       return ((*pos1 == *pos2) && (*pos1  == *pos3));
     }
  }
}
```

move_t Game::last (void) Returns the last move made.

Warning:

Games with no moves yet return an undefined value.

```
{
  TAU_PROFILE("Game::last()", "", TAU_DEFAULT);
```

```
   static move_t mv;

   if (theGame.size() == 1) {
      cout << "-E- No moves made yet.  Cannot get last move." << endl;
      return mv;
   } else {
      //Board b = *theGame.rbegin();
      Board b = theGame.back();
      mv.loc = b.get_move_played();
      mv.color = b.get_color_played();
      if (mv.loc == Board::PASS) { mv.pass = true; };

      if (theGame.size() == 2) {
        mv.newboard = true;
      } else {
        mv.newboard = false;
      }

      mv.setup_phase = false;
      //mv.bsize = this->bsize;
      return mv;
   }
}
```

bool Game::legal (int *x*, int *y*) Is the move legal?

Warning:

Passes are always legal. This function does not accept semantics of "pass"

```
{ return legal(y * Board::BSIZE + x); }
```

bool Game::legal (loc_t *loc*) Tests if a move is legal.

Todo:

Add memoizability–> store vector of legal/not-legal that is updated as moves are made.

```
{
   TAU_PROFILE("Game::legal()", "", TAU_DEFAULT);

   usi_p captures;

  // Only locations on board and not taken already...
  if (!currentBoard->valid_location(loc)) {
```

```
    return false;
  }

  // KO violation checking. Go back to the move before the previous
  // move. It should not be the same board.
  list<Board>::reverse_iterator pos;
  pos=theGame.rbegin();
  if (pos!=theGame.rend()) ++pos;
  // If there are less than two previous boards, can't check for KO
  if (pos!=theGame.rend()) {
     Board b(*currentBoard);
     captures = b.play_move(loc, whose_turn);
     // b is current board with suggested move played and *pos is
     // potential KO violation board (i.e. the same exact board)
     if (*pos == b) return false;


#if (SUPERKO_CHECK == 1)
     // Super KO violation.  Could store a hash to make this quicker, but
     // it's not worth it since I probably won't use this functionality.
     if (super_ko) {
         // Loop through all boards
         while (++pos != theGame.rend()) {
           if (*pos == b) return false;
         }
     }
#endif

  }

#if (SUICIDE_CHECK == 1)
  // Suicide violation
  if (!suicide && (captures.second > 0)) return false;
#endif

  return true;
}
```

void Game::lock (void) Locks the class

```
                     {
   pthread_mutex_lock(&mutex);
}
```

int Game::movenum (void) Tells the current move number

This function calculates this value based on the number of boards in the game.

```
{
   return (theGame.size()-1);
}
```

bool Game::operator!= (const Game & *other*) Inequality operator

```
                                        {
   return (!(*this == other));
}
```

void Game::play_move (loc_t *l*) Play a move.

Parameters:

loc A location to play a move on (must be legal!)

Warning:

Does not do error checking or validity checking

```
{
   TAU_PROFILE("Game::play_move()", "", TAU_DEFAULT);

   Board b;
   pair<usi_t, usi_t> captures;

   captures.first = 0;
   captures.second = 0;

   // Copy the last board onto the end of the list
   b=*currentBoard;
   theGame.push_back(b);
   ++currentBoard;

   //enum_memoize_flag = false;

   if (loc != Board::PASS) {
      captures = currentBoard->play_move(loc, whose_turn);
      if (whose_turn==WHITE) {
         swap(captures.first, captures.second);
      }
   }
   capStones.push_back(captures);

   // Make it the other color's turn
   inv_turn();
}
```

void Game::reset (void) Totally clears and resets the game to initial state.

```
{
   Board b;

   theGame.clear();
   theGame.push_back(b);

   capStones.clear();
   usi_p capturedStones;
   capturedStones.first=0;
   capturedStones.second=0;
   capStones.push_back(capturedStones);

   whose_turn = BLACK;

   //enum_memoize_flag = false;

   currentBoard = theGame.begin();
}
```

void Game::retract (usi_t *num*) Retracts moves.

This function completely destroys all record of the previous num moves. Since the moves/board-states are stored in a list, retracting is a very simple matter.

Parameters:

num The number of moves to retract

```
{
   TAU_PROFILE("Game::retract()", "", TAU_DEFAULT);

   // Protect against retracting past first move.
   if (theGame.size() <= num) {
      num = theGame.size() - 1;
   }

   // Remove last "num" boards
   for (int x=0; x<num; ++x) { theGame.pop_back(); }

   // Remove last "num" captured stones pairs
   for (int x=0; x<num; ++x) { capStones.pop_back(); }

   assert(theGame.size() == capStones.size());

   // Set whose turn it is.
   if (odd(num)) { inv_turn(); }

```

```
    // Set current board iterator as the last board in the list
    currentBoard = theGame.end();
    --currentBoard;
}
```

A.2.9.3 Friends And Related Function Documentation

ostream & operator<< (ostream & *strm*, Game & *aGame*) `[friend]`
Stream operator.

This is used to output the latest state of the game

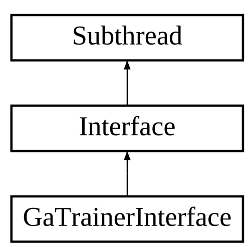
```
                                                 {
    Board b = aGame.get_board();
    strm << b;
    return strm;
}
```

A.2.10 GaTrainerInterface Class Reference

Used to train a GA to work correctly.

`#include <interface.h>`

Inheritance diagram for GaTrainerInterface::



Public Methods

- **GaTrainerInterface** ()

  Constructor.

- ~**GaTrainerInterface** ()

  Destructor.

- void **load** (string fname)

  Loads into memory the training data.

- float **get_percentage** ()

  Figures fraction of correct guesses.

Private Methods

- void **processing** (void)

  Main processing loop.

- void **handle_move** (void)

  Handles modifications to setup board for opponent given the correct move.

- void **init** (void)

  Initializes **Interface** (p. 144).

Private Attributes

- int **totalmoves**

  Moves played.

- int **movesGuessed**

  Moves played correctly.

- list<**move_t**> **movestream**

  The correct moves (from recorded games).

- list<**move_t**>::iterator **movestream_iter**

  Iterator for movestream.

A.2.10.1 Detailed Description

Used to train a GA to work correctly.

A.2.10.2 Member Function Documentation

float GaTrainerInterface::get_percentage () Figures fraction of correct guesses.

This function looks at the number of moves in the game record and the number of moves guessed correctly and calcualtes the fraction of the moves guessed correctly.

Returns:

Fraction of the recorded game moves guessed correctly

```
{
   if (totalmoves == 0) { return static_cast<float>(0); };

   float perc = static_cast<float>(movesGuessed) /
                static_cast<float>(totalmoves);

   LOG("-GAT        -M- Got " << perc << " right.");

   return perc;
}
```

void GaTrainerInterface::init (void) [private] Initializes **Interface** (p. 144).

Precondition:

movestream has been loaded with data via the **load**() (p. 125) function call.

```
{
   // Point to data start
   movestream_iter = movestream.begin();
}
```

void GaTrainerInterface::load (string *fname*) Loads into memory the training data.

The format is a space delimited record with records marked with newline characters PASS MOVE_LOCATION COLOR IGNORE IGNORE BOARDSIZE. all fields are one character except MOVE_LOCATION which is three MOVE_-LOCATION is an offset into a single-dimention array.

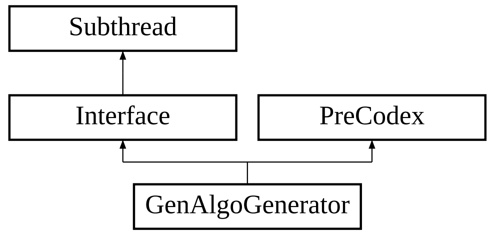
```
{
   ifstream fin(fname.c_str());
   move_t move;


   if (fin) {
     while (!fin.eof()) {
       fin >> move;

       if (move.bsize != Board::BSIZE) {
          cout << "-E- Requested board size and board size in training "
               << "data do not match: " << move.bsize << " and "
               << Board::BSIZE << endl;
          exit(0);
       }

       assert(move.bsize == Board::BSIZE);

       //cout << move.loc << " " << flush;

       movestream.push_back(move);
     }
     fin.close();
     if (fin) {
        cout << "-E- Error closing training data file." << endl;
     }

     // Why?
     movestream.pop_back();

   } else {
      cout << "-E- Error opening training data file." << endl;
   }

   //cout << "Movestream is sized at " << movestream.size() << endl;
}
```

A.2.11 GenAlgoGenerator Class Reference

A genetic algorithm move generator.

`#include <outputgen.h>`

Inheritance diagram for GenAlgoGenerator::



Public Methods

- **GenAlgoGenerator** ()

  Constructor.

- ~**GenAlgoGenerator** ()

  Destructor.

- void **load** (string filename="")

  Loads GA parameters from a file on disk.

- void **printweights** (void)

  Prints the weights for the network to the log file.

- void **decode** (const **chromosome_t** &chrom)

  Decodes chromosome.

- float **get_fitness** (const **chromosome_t** &chrom)

  Objective Function.

- void **summary** (**Population** *newpop)

  Outputs the best chromosome and the weights.

Private Methods

- void **init** (void)

  Sets up agent/generator connections.

- **loc_t get_move** (void)

  Generate a probability board and etc.

- void **processing** (void)

  Main processing loop.

Private Attributes

- int **weights** [SECONDLEVELNODES][MAX_AGENTS]

  Weights to optimize.

- int **secondLevelWeights** [SECONDLEVELNODES]

  Second level weights.

- **Agent*** **theAgents** [MAX_AGENTS]

  Pointers to all agents we use.

- **AgentShell theThreads** [MAX_THREADS]

  The threads for working.

- **ProbBoard results** [MAX_AGENTS]

  Agents put results here.

- **Blackboard bb**

  Information viewable by all agents.

- int **num_agents**

  Number of active agents.

- int **num_threads**

  Number of running threads for agents.

- unsigned int **rndbuf**

  State var used for random number generation.

- bool **weights_loaded**

  Are the weights set yet?

- unsigned int **bits_per_weight**

  Bits used for each weight in the net.

- unsigned int **num_second_level_nodes**

  Number of nodes in level 2.

- unsigned int **total_bits**

  Total number of bits in chromosome.

A.2.11.1 Detailed Description

A genetic algorithm move generator.

A.2.11.2 Member Function Documentation

void GenAlgoGenerator::decode (const chromosome_t & *chrom*) Decodes chromosome.

This function reads chromosome as a string of three integer weights encoded as four bits each (12 bits total)

Author:

Todd Blackman

```
{
   TAU_PROFILE("GenAlgo::decode()", "", TAU_DEFAULT);

   int z=0;
   int tmpweight=0;

   set_chrom_size(total_bits);

   //cout << "-M- " << chrom.size() << " " << total_bits << endl;

   assert(chrom.size() == total_bits);

   LOG("-GAG " << setw(7) << childt << "-M- Loading chromosome: "
               << chrom);

   unsigned int count=0;
   unsigned int count2=0;
   int endOfWeights = num_agents * bits_per_weight * num_second_level_nodes +
                      num_second_level_nodes * bits_per_weight;
   for (int x=0; x<=endOfWeights; ++x) {
      if ((x>0) && (!(x % bits_per_weight))) {

```

```
         if (count < (num_agents * num_second_level_nodes)) {
           weights[count/num_agents][count%num_agents] = tmpweight;
           LOG("            " << "-M- weights[" << count/num_agents << "]["
                              << count%num_agents << "]=" << tmpweight);
         } else if (count < (num_agents*num_second_level_nodes +
                             num_second_level_nodes)) {
           secondLevelWeights[count2] = tmpweight;
           LOG("            " << "-M- secondLevelWeights[" << count2
                              << "]=" << tmpweight);
           count2++;
         }
         count++;

         z=0;
         tmpweight=0;
      }
      tmpweight += chrom[x] * static_cast<int>(pow(2, z));
      ++z;
   }

   // Get extra bits now
   int start = endOfWeights;
   for (int x=0; x<num_agents; ++x) {
      int bits_needed = theAgents[x]->query_bits_needed_from_GA();

      if (bits_needed+endOfWeights <= lchrom) {
        theAgents[x]->send_bits(chrom, start);
        LOG("            " << "-M- Extra bits for agent " << x
                           << " at " << start
                           << " and consisting of " << bits_needed);
        start += bits_needed;
      } else {
        cerr << "-E- Bit count mismatch." << endl;
        exit(0);
      }
   }

  weights_loaded = true;
}
```

float GenAlgoGenerator::get_fitness (const chromosome_t & *chrom*) `[virtual]` Objective Function.

This objective function is exceedingly complex though one cannot tell just by looking here. This function creates a moderator with two players, the GA player and a GA-trainer player. This is then allowed to run until the trainer exhausts its series of moves to play.

Returns:

Fitness value

Reimplemented from **PreCodex** (p. 155).

```
{
    TAU_PROFILE("GenAlgo::get_fitness()", "", TAU_DEFAULT);

    float fitness;

    //LOG("-GAG " << setw(7) << childt << "-M- Entered get_fitness().");

    //Connect the trainer and trainee with a moderator
    Moderator<GenAlgoGenerator, GaTrainerInterface> trainpair;

    cerr << "." << flush;

    // Setup the trainer and trainee (Do we need this at all?)
    //Interface *trainee;
    GenAlgoGenerator *trainee;
    //Interface *trainer;
    GaTrainerInterface *trainer;

    trainee = static_cast<GenAlgoGenerator *>(trainpair.get_I0());
    trainer = static_cast<GaTrainerInterface *>(trainpair.get_I1());

    // Adjust weights based on chromosome
    trainee->decode(chrom);

    // prepare the trainer
    //static_cast<GaTrainerInterface *>(trainer)->load(".test.dat");
    static_cast<GaTrainerInterface *>(trainer)->load(Ga::TRAIN_FILE);

    LOG("-GAG " << setw(7) << childt
                << "-M- About to call mainloop from get_fitness().");

    // Run moderator till trainer says done.
    trainpair.mainloop();

    LOG("-GAG       -M- Done with mainloop in fitness-finding function.");

    // Get stat from trainer interface.
    fitness = static_cast<GaTrainerInterface *>(trainer)->get_percentage();

    LOG("-GAG       -M- Fitness has been calculated.");

    // Return percentage of moves correctly guessed.
    return fitness;
}
```

loc_t GenAlgoGenerator::get_move (void) `[private]` Generate a probability board and etc.

This function calls all agents to the task of creating a probability board. This function then proceeds to sum these boards as defined by the genetic algorithm weight parameters.

```
{
    TAU_PROFILE("GenAlgo::get_move()", "", TAU_DEFAULT);

    loc_t loc;
    ProbBoard pbtmp;
    ProbBoard pbresult;
    msg_t msg;

    //LOG("-GAG " << setw(7) << childt << "-M- Entered get_move()");

    // Fill up as many threads as we have.
    for (int x=0; x<num_threads; x++) {
       assert(theAgents[x] != NULL);

       // Load the next agent
       msg.id = LOAD;
       msg.data = static_cast<void *>(theAgents[x]);
       theThreads[x].send_msg(msg);

       // Update the agent
       msg.id = UPDATE;
       //pthread_mutex_lock(&update_mutex);
       msg.data = static_cast<void *>(gptr);
       //pthread_mutex_unlock(&update_mutex);
       theThreads[x].send_msg(msg);
    }

    int count=0;
    while(count < num_threads) {
       msg = theThreads[count].get_msg_nb();
       if (msg.id == FINISHED) count++;
    }

    //LOG("-GAG " << setw(7) << childt << "-M- Got all finished messages.");

    // Compute probablity board
    pbresult.clear();
    for (unsigned int lcv=0; lcv<num_second_level_nodes; ++lcv) {
      pbtmp.clear();
      // Calcualte results second level results.
      for (int x=0; x<num_agents; ++x) {
         pbtmp += results[x] * weights[lcv][x];

         //cout << "results[" << x << "]=" << endl << results[x]
         //     << endl;
```

```
         //cout << "weights[" << lcv << "][" << x << "]="
         //     << weights[lcv][x] << endl;
         //cout << "pbtmp = " << endl << pbtmp << endl;
         //cout << "^^^^^^^^^^" << endl;

      }
      pbtmp.normalize();
      //cout << "pbtmp = " << pbtmp << endl;
      //cout << "=======" << endl;
      pbresult += pbtmp * secondLevelWeights[lcv];

         //cout << "secondLevelWeights[" << lcv << "]="
         //     << secondLevelWeights[lcv] << endl;
         //cout << "pbresult = " << pbresult << endl;

    }
    pbresult.normalize();

    //cout << "pbresult = " << pbresult << endl;
    //cout << "---------------------------------------------" << endl;

#if SPIN
    loc = pbresult.spin();
    assert(false); // SPIN shouldn't be used unless you really want to
#else
    loc = pbresult.maxloc();
#endif

    if (loc == (Board::BSIZE * Board::BSIZE)) {
       loc = Board::PASS;
    }

    assert((loc <= (Board::BSIZE * Board::BSIZE)) || (loc == Board::PASS));

    return loc;
}
```

void GenAlgoGenerator::load (string *filename* = "") Loads GA parameters from a file on disk.

Author:

Todd Blackman

```
                                            {
   ifstream fin;
   chromosome_t tmpchrome;
   unsigned int chrom_length;
   char c;
```

```

   if (name == "") name = Ga::BEST_FILENAME_OUT;

   fin.open(name.c_str());

   if (fin) {
      fin >> chrom_length;

      // Check that datafile has correct cromosome length
      if (chrom_length != total_bits) {
          cerr << "-E- datafile chromosome length doesn't match: ";
          cerr << total_bits << "!=" << chrom_length << endl;
          LOG("-GAG       -E- Invalid chromosome_t length read");
          return;
      }

      while (fin) {
         fin.get(c);
         if (c == '1') tmpchrome.push_back(true);
         if (c == '0') tmpchrome.push_back(false);
      }
      fin.close();

      if (tmpchrome.size() != total_bits) {
          cerr << "-E- datafile chromosome length doesn't match: ";
          cerr << total_bits << "!=" << tmpchrome.size() << endl;
          LOG("-GAG       -E- Invalid chromosome_t length read");
          return;
      }


  } else {
     LOG("-GAG      -E- Cannot load Genetic Algorithm data from disk.");
  }

  decode(tmpchrome);

}
```

A.2.12 global_data_t Struct Reference

global data structure.

`#include <exodus.h>`

Public Attributes

- bool **welcome**

Show welcome screen to user.

- bool **train**

  Train GA or not.

- int **verbosity**

  How much output to output (unused).

- char∗ **resume**

  File name to resume a GA run.

- bool **version**

  Show version number or not.

- bool **help**

  Show help message or not.

- bool **reg_on**

  Run regression or not.

- **color_t my_color**

  Human player's color (untested).

- char∗ **handicap_placement**

  Where to place the handicap stones.

A.2.12.1 Detailed Description

global data structure.

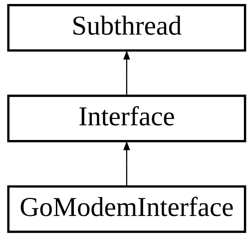
Make this as small as possible

A.2.13 GoModemInterface Class Reference

Go modem interface.

`#include <interface.h>`

Inheritance diagram for GoModemInterface::



Public Methods

- **GoModemInterface** ()

  Constructor.

A.2.13.1 Detailed Description

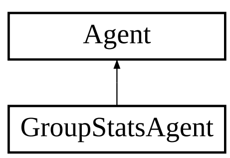
Go modem interface.

A.2.14 GroupStatsAgent Class Reference

Agent (p. 73) to calculate group information.

`#include <agent.h>`

Inheritance diagram for GroupStatsAgent::



Public Methods

- **GroupStatsAgent** ()

  Constructor.

- ~**GroupStatsAgent** ()

  Destructor.

- void **force** (void)

  Force the agent to move.

- void **update** (**Game** *)

  Updates the agent with the latest state of the game.

- bool **dowork** (void)

  work thread. Kills groups.

- void **notify** (void *)

  Tell the agent something.

- unsigned int **query_bits_needed_from_GA** (void)

  Asks the agent for the number of bits it needs from GA.

- void **send_bits** (**chromosome_t** chrom, int start)

Private Methods

- void **printScratch** (void)

  Prints the group scratchpad to STDOUT.

- void **recurse** (**Stone** goban[ ], int loc, int gnum)

  Recursive function to label a group by number.

Private Attributes

- int **scratch** [19 *19]

  Holds a bitmap that shows the group numbers. Each group is uniquely identified by a number.

- int **numgroups**

  Number of groups.

- bool **dead** [MAXGROUPS]

  Which groups are dead and which alive.

- **color_t grpcolor** [MAXGROUPS]

  Color of the group.

- unsigned int **gsize** [MAXGROUPS]

  Size of the group.

- unsigned int **liberties** [MAXGROUPS]

  Number of liberties.

- bool **liberty_locations** [MAXGROUPS][19 *19]

  Liberty locations as a bitmap.

- list<**loc_t**> **liberty_locations_list** [MAXGROUPS]

  Liberty locations as a list of locations.

A.2.14.1 Detailed Description

Agent (p. 73) to calculate group information.

A.2.14.2 Member Function Documentation

void GroupStatsAgent::send_bits (chromosome_t *chrom*, int *start*) [virtual] Sends to this agent the bits it needs from the GA

Reimplemented from **Agent** (p. 75).

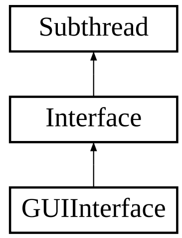
```
{
    //int end = start + query_bits_needed_from_GA();

}
```

A.2.15 GUIInterface Class Reference

Graphical User **Interface** (p. 144).

#include <interface.h>

Inheritance diagram for GUIInterface::



Public Methods

- **GUIInterface** (**usi_t** size=19)

  Constructor.

- **GUIInterface** (string thepath, **usi_t** size=19)

  Constructor.

- ~**GUIInterface** ()

  Destructor.

Static Public Attributes

- string **GPATH**

  Path to GUI frontend.

Private Methods

- **loc_t get_move** (void)

  Gets move from GUI.

- void **send_board** (**Board** b, **color_t** whose_turn)

  Sends the current board to the interface from engine.

- void **init** (void)

  Forks off the gui.

- void **figure_path** (void)

  Sets path to gui.

- void **processing** (void)

  Main logic loop of the interface.

Private Attributes

- int **m2s** [2]

  Master to slave flow (interface to outside).

- int **s2m** [2]

  Slave to master flow (outside to interface).

- int **pid**

  Child Process ID.

- **usi_t bsize**

  Board (p. 78) *size.*

- string **path**

  Path to gui program.

Static Private Attributes

- const int **READ** = 0

  Constant.

- const int **WRITE** = 1

  Constant.

A.2.15.1 Detailed Description

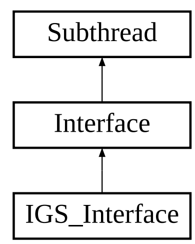
Graphical User **Interface** (p. 144).

A.2.16 IGS_Interface Class Reference

Internet Go Server (IGS) **Interface** (p. 144).

`#include <interface.h>`

Inheritance diagram for IGS_Interface::



Public Methods

- **IGS_Interface** ()

  Constructor.

- ~**IGS_Interface** ()

  Destructor.

Private Methods

- void **setup** (void)

  Initializes the class.

Private Attributes

- string **host1**

  First IGS server to try.

- string **host2**

  Second IGS server to try.

- **usi_t port1**

  Port on first host.

- **usi_t port2**

  Port on second host.

- sockaddr **my_addr**

  Local IP address.

- int **sfd**

  File descriptor for socket connection.

A.2.16.1 Detailed Description

Internet Go Server (IGS) **Interface** (p. 144).

Warning:
  This class is a stub.

A.2.17 Individual Struct Reference

An individual in a population of a GA.

```
#include <ga.h>
```

Public Methods

- bool **operator==** (const Individual &) const

  Equality operator.

- bool **operator!=** (const Individual &) const

  Inequality operator.

- Individual **operator=** (Individual)

  Assignment operator.

Public Attributes

- **chromosome_t chrom**

  The chromosome that represents this individual.

- float **ofitness**

  Original fitness.

- float **fitness**

  Fitness after scaling.

- **usi_t parent1**

  First parent chromosome.

- **usi_t parent2**

  Second parent chromosome.

- **usi_t xsite**

  Site of crossover.

A.2.17.1 Detailed Description

An individual in a population of a GA.

A.2.17.2 Member Function Documentation

bool Individual::operator== (const Individual & *other*) const Equality operator.

Warning:

Not used for now. This will be useful when the algorithm is multi-threaded which it currently isn't.

void Ga::interrupt() { pthread_mutex_lock(&interrupt_watcher); stop=true; pthread_mutex_unlock(&interrupt_watcher); }

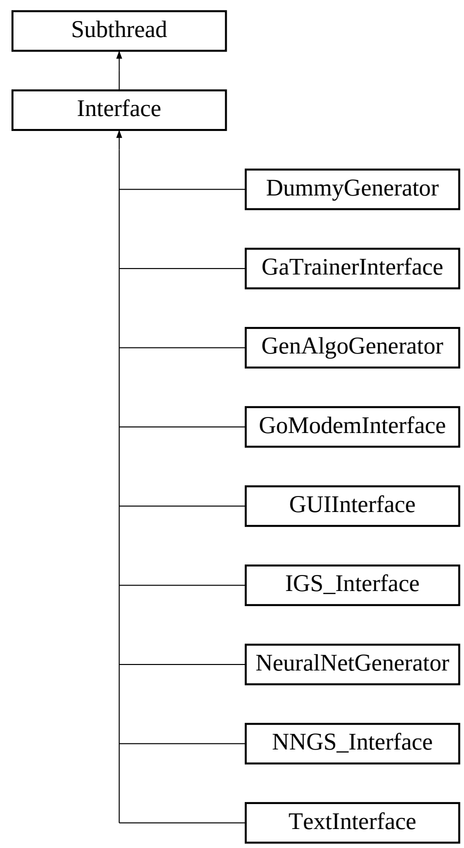
```
{
  return(this->chrom == other.chrom);
}
```

A.2.18 Interface Class Reference

The interface between a move generator (outside) and the inside of the program.

`#include <interface.h>`

Inheritance diagram for Interface::



Public Methods

- **Interface** ()

  Constructor.

- ~**Interface** ()

  Destructor.

- void **set_my_turn_on** (void)

  Moderator (p. 146) *class uses these to control who is "active" meaning.*

- bool **get_made_a_move** (void)

Moderator (p. 146) *class uses these to control who is "active" meaning "whose turn is it?".*

- void **set_my_color** (const **color_t** col)

Protected Attributes

- bool **my_turn**

  Is it my turn?

- bool **made_a_move**

  Have I made my move for this round?

- **Game∗ gptr**

  Points to current **Game** (p. 112).

- **color_t my_color**

  My color.

- **color_t their_color**

  Opponent's color.

A.2.18.1 Detailed Description

The interface between a move generator (outside) and the inside of the program.

Warning:

This is an abstract class

set_die_ptr of the Subthread class must be set before using any interface. In addition to this, gptr and resign_ptr need to be set before subthread's **start**() (p. 165) function is called.

A.2.18.2 Member Function Documentation

void Interface::set_my_color (const color_t *col*) `[inline]` Sets interface's color

```
{ my_color = col;
  their_color = INV(col); };
```

A.2.19 Moderator Class Template Reference

Encapsulates two interfaces and has them play together.

Public Methods

- **Moderator** ()

  Constructor.

- ~**Moderator** ()

  Destructor.

- **Interface∗ get_I0** (void)

  Retrieves the first interface.

- **Interface∗ get_I1** (void)

  Retrieves the second interface.

- **Game∗ get_game** (void)

  Retrieves the game.

- void **mainloop** (void)

  Lets the interfaces play with each other.

- void **swap_interfaces** (void)

  Swaps semantics of I0 and I1.

Private Attributes

- I0_t∗ **I0**

  First **Interface** (p. 144) *(0).*

- I1_t∗ **I1**

  Second **Interface** (p. 144) *(1).*

- bool **whose_turn**

  Which interface's turn is it?

- **Game theGame**

  The game.

A.2.19.1 Detailed Description

template<class I0_t, class I1_t> class Moderator Encapsulates two interfaces and has them play together.

The first interface, I0, Is the black player and thus I1 receives the handicap by definition. To alter this, one needs only to swap the two interfaces using the method **swap_interfaces**() (p. 146).

Warning:

The functions **get_I0**() (p. 146), **get_I1**() (p. 146), and **get_game**() (p. 146) should be used wisely. They are sources of error and potential faults, but I'll trust myself and potential future programmers to not screw the semantics up like making two interfaces think the board is a different size than it is.

Todo:

Add time-keeping code.

A.2.19.2 Constructor & Destructor Documentation

template<class I0_t, class I1_t> Moderator< I0_t, I1_t >::Moderator<I0_t, I1_t> () Constructor.

Warning:

I0 always goes first

```
{

   I0 = new(I0_t);
   I1 = new(I1_t);
   if ((!I0) || (!I1)) {
      LOG("-MOD       -E- Interface memory allocation failed.");
      cerr << "-E- Interface memory allocation failed.";
      exit(1);
   }

   // Set Game class for two opponents
   msg_t msg;
   msg.id = SET_GAME_PTR;
```

```
    msg.data = static_cast<void *>(&theGame);
    I0->send_msg(msg);
    I1->send_msg(msg);
}
```

A.2.20 move_t Struct Reference

A single move on the goban.

```
#include <move.h>
```

Public Methods

- bool **operator==** (move_t &m)

  equality operator.

- move_t **operator=** (move_t other)

  assignment operator.

- void **regression** (void)

  Unused.

Public Attributes

- bool **pass**

  Is the move a pass?

- **loc_t loc**

  Where the move is played.

- **color_t color**

  Color of the move played.

- bool **newboard**

  Is this the first of a new board?

- bool **setup_phase**

  Still setting up handicaps?

- int **bsize**

  Size of the board.

A.2.20.1 Detailed Description

A single move on the goban.

A.2.21 msg_t Struct Reference

A message to or from a thread.

`#include <subthread.h>`

Public Methods

- msg_t **operator=** (msg_t)

  Assignment operator.

Public Attributes

- **msg_id_t id**

  Message ID (type).

- void* **data**

  Message payload.

A.2.21.1 Detailed Description

A message to or from a thread.

A.2.21.2 Member Function Documentation

msg_t msg_t::operator= (msg_t *tmpmsg*) Assignment operator.

equality operator for a message

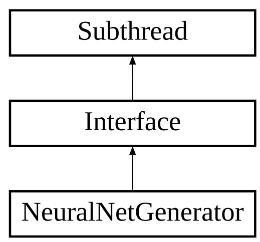
```
                                      {
     this->id = tmpmsg.id;
     this->data = tmpmsg.data;
     return *this;
}
```

A.2.22 NeuralNetGenerator Class Reference

A Neural Network move generator.

#include <outputgen.h>

Inheritance diagram for NeuralNetGenerator::



Public Methods

- **NeuralNetGenerator** ()

  Constructor.

A.2.22.1 Detailed Description

A Neural Network move generator.

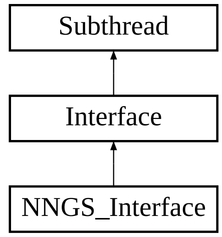
Warning:
This is a STUB

A.2.23 NNGS_Interface Class Reference

No Name Go Server **Interface** (p. 144).

#include <interface.h>

Inheritance diagram for NNGS_Interface::



Public Methods

- **NNGS_Interface** ()

  Constructor.

A.2.23.1 Detailed Description

No Name Go Server **Interface** (p. 144).

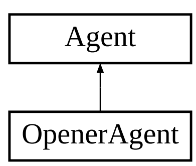
Warning:
This class is a STUB.

A.2.24 OpenerAgent Class Reference

Suggests good opening moves.

`#include <agent.h>`

Inheritance diagram for OpenerAgent::



Public Methods

- **OpenerAgent** ()

  Constructor.

- **~OpenerAgent** ()

  Destructor.

- void **force** (void)

    Force agent to move.

- void **update** (**Game** ∗)

    Update the agent with the current game.

- bool **dowork** (void)

    work thread.

- void **notify** (void ∗)

    Tell the agent something.

- unsigned int **query_bits_needed_from_GA** (void)

    Ask the agent how many bits it needs in GA.

- void **send_bits** (**chromosome_t** chrom, int start)

Private Attributes

- **ProbBoard pb19**

    Stores choices for 19x19 board.

- **ProbBoard pb17**

    Stores choices for 17x17 board.

- **ProbBoard pb9**

    Stores choices for 9x9 board.

A.2.24.1 Detailed Description

Suggests good opening moves.

A.2.24.2 Member Function Documentation

void OpenerAgent::send_bits (chromosome_t *chrom*, int *start*) `[virtual]` Sends to this agent the bits it needs from the GA

Reimplemented from **Agent** (p. 75).

```
{
    //int end = start + query_bits_needed_from_GA();
}
```

A.2.25 Population Struct Reference

A single population within a GA.

```
#include <ga.h>
```

Public Methods

- bool **operator==** (const Population &) const

  Equality operator.

- bool **operator!=** (const Population &) const

  Inequality operator.

- Population **operator=** (Population)

  Population assignment operator.

Public Attributes

- vector<**Individual**> **individuals**

  The individuals in the chromosome.

- float **sumfitness**

  Sum of all fitness values in this generation.

- float **avg**

  Average fitness in this generation.

- float **max**

  Maximum fitness in this generation.

- float **min**

  Minimum fitness in this generation.

- float **stdev**

  Standard deviation in this generation.

- float **var**

  Variance deviation in this generation.

- int **whichmax**

  Which in population is max.

- int **whichmin**

  Which in population is min.

Friends

- ostream& **operator**<< (ostream &strm, const Population &pop)

  Population output operator.

A.2.25.1 Detailed Description

A single population within a GA.

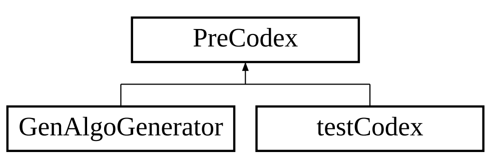
Every GA has two populations: An old one and a new one.

A.2.26 PreCodex Class Reference

Allows other classes to profide a fitness function.

`#include <gafunc.h>`

Inheritance diagram for PreCodex::



Public Methods

- void **set_chrom_size** (int s)

Sets the chromosome size to s.

- int **get_chrom_size** (void)

  Gets the current chromosome size.

- virtual float **get_fitness** (const **chromosome_t** &chrom)=0

  Decoder and Objective function.

- virtual void **summary** (**Population** *newpop)=0

  Outputs testing results after training is done.

Protected Attributes

- int **lchrom**

  Length of chromosome in bits (alleles).

A.2.26.1 Detailed Description

Allows other classes to profide a fitness function.

A.2.27 ProbBoard Class Reference

Agent (p. 73)'s probability output board.

```
#include <probboard.h>
```

Public Methods

- **ProbBoard** ()

  Constructor.

- ~**ProbBoard** ()

  Destructor.

- void **set_val** (int offset, float value)

  Sets weight.

- float **get_val** (int offset)

  Gets weight.

- void **normalize** (void)

  Normalizes the weights.

- int **spin** (void)

  Chooses a random offset in probability board based on probabilties.

- **loc_t maxloc** (void)

  Choose the location with the highest value.

- void **clear** (void)

  Clears board.

- ProbBoard **operator=** (ProbBoard)

  Assignment operator.

- bool **operator==** (ProbBoard)

  equality overloaded operator.

- bool **operator!=** (ProbBoard)

  Inequality operator.

- bool **operator+=** (ProbBoard)

  Addition assignment operator.

- ProbBoard **operator** * (float) const

  Multiplication operator.

- ProbBoard **operator+** (ProbBoard)

  Addition operator.

- float& **operator[]** (**loc_t** location)

  Offset and Array-use operator.

Private Attributes

- int **actualSize**

  Size of internal array.

- float **internal_board** [19 *19+1]

  Single dimention array.

- unsigned int **rndbuf**

  Seed for random number generator.

Friends

- ostream& **operator**<< (ostream &strm, ProbBoard &aBoard)

  Stream operator.

A.2.27.1 Detailed Description

Agent (p. 73)'s probability output board.

A.2.27.2 Member Function Documentation

loc_t ProbBoard::maxloc (void) Choose the location with the highest value.

Heuristic would have the FIRST of any tie values chossen.

```
{
   TAU_PROFILE("ProbBoard::maxloc()", "", TAU_DEFAULT);
   float max=0;
   loc_t loc=Board::PASS;

   for(loc_t j=0; j<actualSize; j++) {
      if (internal_board[j] > max) {
         max = internal_board[j];
         loc = j;
      }
   }

   return loc;
}
```

void ProbBoard::normalize (void) Normalizes the weights.

Normalization makes the sum of all weights equal to 1.

```
{
   TAU_PROFILE("ProbBoard::normalize()", "", TAU_DEFAULT);

   float sum = 0.0;

   // Find the sum
   for (int lcv = 0; lcv < actualSize; ++lcv) sum += internal_board[lcv];

   // Convert board into percentage board (normalize)
   if (sum != static_cast<float>(0.0)) {
     for (int lcv = 0; lcv < actualSize; ++lcv)
        internal_board[lcv] = internal_board[lcv] / sum;
   }
}
```

int ProbBoard::spin (void) Chooses a random offset in probability board based on probabilties.

Precondition:

The sum of the probibility board locations is very close to 1 or else the sum is zero.

Returns:

The chosen location

```
{
   TAU_PROFILE("ProbBoard::spin()", "", TAU_DEFAULT);
   float target;
   float current = 0.0;
   int j = 0;

   // Pick a number between zero and one
   target = static_cast<float>(rand_r(&rndbuf)) / static_cast<float>(INT_MAX);

   assert(target >= 0);
   assert(target <= 1.0);

   // todo: Rewrite (simplify, it's easy)
   //while (((current < target) && (j < actualSize))  ||
   //         ((internal_board[j] == 0) && (j < actualSize))) {
   while ((j < actualSize) &&
          ((current < target) || ((j>0) && (internal_board[j-1] == 0)))) {
      current += internal_board[j++];
   }
```

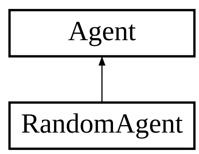
```
   j--;

   //if (j == actualSize) { --j; }

   assert(j < actualSize);

   return j;
}
```

A.2.28 RandomAgent Class Reference

Suggests random legal moves.

#include <agent.h>

Inheritance diagram for RandomAgent::



Public Methods

- **RandomAgent** ()

  Constructor.

- ~**RandomAgent** ()

  Destructor.

- void **force** (void)

  Force the Random agent to make its move.

- void **update** (**Game** *)

  Updates the game for the agent. Refresh agent with a new game state.

- bool **dowork** (void)

  work thread.

- void **notify** (void *)

  Tell the agent something.

- unsigned int **query_bits_needed_from_GA** (void)

    Asks the agent how many bits it needs in the GA.

- void **send_bits** (**chromosome_t** chrom, int start)

    Sends to this agent the bits it needs from the GA.

A.2.28.1 Detailed Description

Suggests random legal moves.

A.2.28.2 Member Function Documentation

void RandomAgent::force (void) `[virtual]` Force the Random agent to make its move.

This function is just here for completeness.

Reimplemented from **Agent** (p. 74).

```
0054 {};
```

unsigned int RandomAgent::query_bits_needed_from_GA (void) `[virtual]` Asks the agent how many bits it needs in the GA.

Returns:

Number of bits needed in GA chromosome

Reimplemented from **Agent** (p. 75).

```
0069 { return 7; }
```

A.2.29 Stone Class Reference

Defines a point (stone) on the board.

`#include <stone.h>`

Public Methods

- **Stone** ()

  Constructor I.

- **Stone** (int)

  Constructor II.

- **Stone** (const Stone &other)

  Copy Constructor III.

- bool **white** (void) const

  Is stone white?

- bool **black** (void) const

  Is stone black?

- bool **empty** (void) const

  Is there a stone?

- bool **notempty** (void) const

  Is there no stone?

- bool **notblack** (void) const

  Is there a stone that isn't black (empty or white)?

- bool **notwhite** (void) const

  Is there a stone that isn't white (empty or black)?

- bool **notleft** (void) const

  Not the leftmost column.

- bool **notright** (void) const

  Not the rightmost column.

- bool **nottop** (void) const

  Not the topmost column.

- bool **notbottom** (void) const

Not the bottommost column.

- int **getrow** (void) const

  Get row stone is in.

- int **getcol** (void) const

  Get column stone is in.

- int **lastrow** (void) const

  Is stone in last row.

- int **lastcol** (void) const

  Is stone in last column.

- **color_t getcolor** (void) const

  Gets the color of the stone.

- void **setrow** (**stone_t**)

  Set stone's row.

- void **setcol** (**stone_t**)

  Set stone's column.

- void **setcolor** (**color_t**)

  Set stone's color.

- void **setlastrow** (void)

  Stone is in last row.

- void **setlastcol** (void)

  Stone is in last column.

- void **clearlastrow** (void)

  Stone is not in last row.

- void **clearlastcol** (void)

  Stone is not in last column.

- void **clear** (void)

  clear stone's bits.

- char **stoneOut** (void)

  text board output.

- bool **operator==** (const Stone &) const

  Stones are the same color (or empty).

- bool **operator!=** (const Stone &) const

  Stones are not the same color (or empty).

- Stone **operator=** (Stone)

  Overload the assignment operator.

Private Attributes

- **stone_t theStone**

  Bit-map representing a stone.

Static Private Attributes

- const **stone_t WHITE_BIT** = 0x0001

  0000 0000 0000 0001.

- const **stone_t BLACK_BIT** = 0x0002

  0000 0000 0000 0010.

- const **stone_t BW_BITS** = 0x0003

  0000 0000 0000 0011.

- const **stone_t ROW_BITS** = 0x007C

  0000 0000 0111 1100.

- const **stone_t COL_BITS** = 0x0F80

  0000 1111 1000 0000.

- const **stone_t LROW_BIT** = 0x1000

  0001 0000 0000 0000.

- const **stone_t LCOL_BIT** = 0x2000

  0010 0000 0000 0000.

Friends

- ostream& **operator**<< (ostream &strm, Stone &aStone)

  Overload the printing operator.

A.2.29.1 Detailed Description

Defines a point (stone) on the board.

A.2.29.2 Member Function Documentation

bool Stone::operator!= (const Stone & *other*) const Stones are not the same color (or empty).

Warning:

The stones are compared via color only. The other bits are ignored

```
                                                   {
  return ((other.theStone & BW_BITS) !=
          (this->theStone & BW_BITS)) ? true : false;
}
```

bool Stone::operator== (const Stone & *other*) const Stones are the same color (or empty).

Warning:

The stones are compared via color only. The other bits are ignored

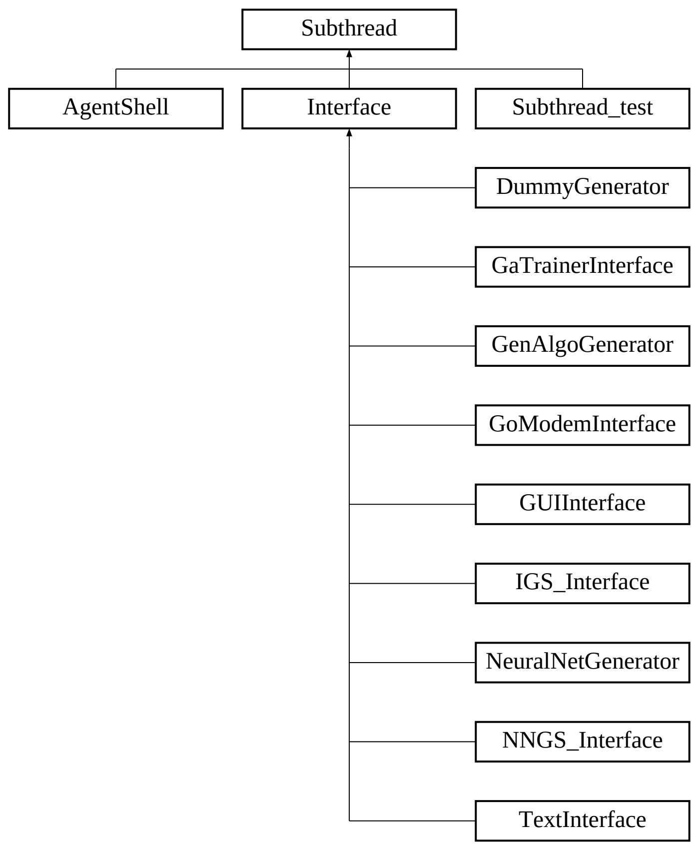
```
                                                   {
  return ((other.theStone & BW_BITS) ==
          (this->theStone & BW_BITS)) ? true : false;
}
```

A.2.30 Subthread Class Reference

Defines a sub-thread.

`#include <subthread.h>`

Inheritance diagram for Subthread::



Public Methods

- **Subthread** ()

  Constructor.

- virtual ~**Subthread** ()

  Destructor.

- void **start** (void)

  Starts the thread running.

- void **kill** (void)

Stops the thread from running (kills it).

- void **send_msg** (**msg_t** msg)

  queues a message for this thread.

- **msg_t get_msg_nb** (void)

  Gets a message and returns if not there.

- void **join** (void)

  Do a thread join on this thread.

Public Attributes

- pthread_t **childt**

  Processing thread.

Protected Methods

- void **inside_send_msg** (**msg_t** msg)

  Allows thread to send out a message.

- **msg_t inside_get_msg_nb** (void)

  Allows thread to get a message (non-blocking).

- **msg_t inside_get_msg_b** (void)

  Allows thread to get a message (blocking).

- virtual void **processing** (void)=0

  Main logic loop of the interface.

- void **tell_message** (int, char *)

  Used to log message types (transform from number to enum text).

Protected Attributes

- pthread_cond_t **block_cond**

  Condition variable for blocking receiving of messages.

- pthread_mutex_t **message_queue_mutex**

  Mutex for both queues.

- queue<**msg_t**> **tothreadq**

  Sending and receiving queues.

- queue<**msg_t**> **fromthreadq**

  Sending and receiving queues.

Friends

- void* **CALL_processing** (void *tmp_obj)

  Calls processing thread.

A.2.30.1 Detailed Description

Defines a sub-thread.

This class supplies a child class with the ability to run in the background as a separate thread.

This is a virtual class, but for all classes that inherit
from this class, one must be sure to call the set_die_ptr() function before calling the function **start**() (p. 165).

A.2.30.2 Constructor & Destructor Documentation

Subthread::~Subthread () `[virtual]` Destructor.

Warning:
This function does not destroy the subthread. This is because the user will nicely kill the subthread via a QUIT message.

```
{
   pthread_mutex_destroy(&message_queue_mutex);
   pthread_cond_destroy(&block_cond);
}
```

A.2.30.3 Member Function Documentation

void Subthread::join (void) Do a thread join on this thread.

Joins this thread

This function joins the thread that this class created before.

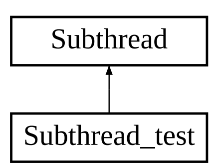
```
                       {
   assert(childt != 0);
   if (pthread_join(childt, NULL)) {
      cout << "-E- Error in joining child." << endl;
      exit(823);
   } else {
     pthread_mutex_lock(&log_mutex);
     thread_count--;
     pthread_mutex_unlock(&log_mutex);
     childt = static_cast<pthread_t>(0);
   }
}
```

A.2.31 Subthread_test Class Reference

For debugging.

`#include <subthread.h>`

Inheritance diagram for Subthread_test::



Public Methods

- ~**Subthread_test** ()

  Descructor.

- void **regression** (void)

Regression.

Protected Methods

- void **processing** (void)
  Main logic loop of the interface.

A.2.31.1 Detailed Description

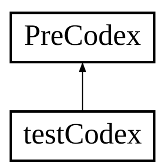
For debugging.

A.2.32 testCodex Class Reference

A testing fitness function provider.

`#include <gafunc.h>`

Inheritance diagram for testCodex::



Public Methods

- **testCodex** ()
  Constructor.
- float **get_fitness** (const **chromosome_t** &chrom)
  Finds the fitness (objective) function value.
- void **summary** (**Population** *newpop)
  Outputs testing results after training is done.

A.2.32.1 Detailed Description

A testing fitness function provider.

A.2.32.2 Member Function Documentation

float testCodex::get_fitness (const chromosome_t & *chrom*) [virtual]
Finds the fitness (objective) function value.

Parameters:

chrom A chromosome to decode and then find the fitness of

Returns:

fitness value

Reimplemented from **PreCodex** (p. 155).

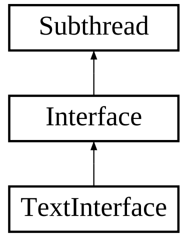
```
{
   float res;

   // Decode
   int sum = 0;
   for(int x=0; x<lchrom; ++x) {
      sum += static_cast<long int>(pow(2, x) *
             static_cast<long int>(chrom[x]));
   }

   // Calculate fitness
   res = static_cast<float>(sum) /
         static_cast<float>(pow(2, lchrom) - 1.0);

   res *= 10.0;

   assert(res <= 10.0);

   if (res < 0) res = 0;

   return res;
}
```

A.2.33 TextInterface Class Reference

Text **Interface** (p. 144).

#include <interface.h>

Inheritance diagram for TextInterface::



Public Methods

- **TextInterface** ()

  Constructor.

Private Methods

- **loc_t get_user_input** ()

  Gets user input.

- void **processing** (void)

  Main processing of interface.

Private Attributes

- string **msg**

  Text message before asking user for input.

- string **prompt**

  Prompt for user input.

A.2.33.1 Detailed Description

Text **Interface** (p. 144).

A.2.33.2 Member Function Documentation

void TextInterface::processing (void) `[private, virtual]` Main processing of interface.

This function, which the base class **Interface** (p. 144) defines as a pure virtual function, provides the main bulk of the logic of the interface.

Warning:

The interface may have outside signals that need to be seen. Thus, the function should finish (return) periodically. It will then be recalled as it is inside an infinite loop that checks for signals then calls this function again.

Reimplemented from **Subthread** (p. 166).

```
{
   loc_t loc;
   Board tb;
   msg_t packetmsg;


   while (true) {

      if (my_turn) {
         packetmsg = inside_get_msg_nb();
      }  else {
         // Nothing to do here but wait for a message (block).
         packetmsg = inside_get_msg_b();
      }

      if (packetmsg.id == QUIT) {
         LOG("-TIN " << setw(7) << childt
                     << "-M- Text Interface exiting.");
         pthread_exit(0);
      } else if (packetmsg.id == RESIGN) {
         LOG("-TIN " << setw(7) << childt
                     << "-M- Got RESIGN message.");
         pthread_exit(0);
      } else if (packetmsg.id == FORCE ) {
         LOG("-TIN " << setw(7) << childt
                     << "-W- Forcing a move not allowed.");
      } else if (packetmsg.id == SET_GAME_PTR) {
         gptr = static_cast<Game *>(packetmsg.data);
      } else if (packetmsg.id == TURN) {
         my_turn = true;
      } else if (my_turn) {

        // Redisplay board (even if board hasn't changed)

        // Get a move and play it.
        loc = get_user_input();
        while (!gptr->legal(loc) && (loc != Board::PASS)) {
           cerr << "-W- That was an illegal move." << endl;
           loc = get_user_input();
```

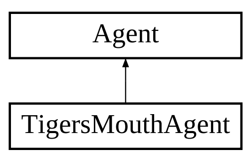
```
        }
        gptr->play_move(loc);


        // Redisplay board
        tb = gptr->get_board();
        cerr << tb << endl;

        my_turn = false;


        // Tell moderator that I'm done.
        packetmsg.id = TURN;
        inside_send_msg(packetmsg);

      }
   }
}
```

A.2.34 TigersMouthAgent Class Reference

Tries to make tiger's mouths.

`#include <agent.h>`

Inheritance diagram for TigersMouthAgent::



Public Methods

- **TigersMouthAgent** ()

  Constructor.

- ~**TigersMouthAgent** ()

  Destructor.

- void **force** (void)

  Force the agent to make its move.

- void **update** (**Game** ∗)

Update this agent with the latest state of the game.

- bool **dowork** (void)

  Work thread.

- void **notify** (void ∗)

  Tell the agent something.

- unsigned int **query_bits_needed_from_GA** (void)

  Tells how many bits this agent needs from the GA.

- void **send_bits** (**chromosome_t** chrom, int start)

  Sends to this agent the bits it needs from the GA.

Private Methods

- bool **findtiger** (**loc_t** loc, **Board** &b)

  Find tiger's mouths.

A.2.34.1 Detailed Description

Tries to make tiger's mouths.

A.2.34.2 Member Function Documentation

bool TigersMouthAgent::findtiger (loc_t *loc*, Board & *b*) `[private]`
Find tiger's mouths.

This function takes the current board, and a location on the board. It returns true if by playing at this location the player would make at least one tiger's eye.

Parameters:

loc Location to check

b The board to consult

```
{
        Stone *stones = b.get_goban();

```

```
    // . ? .
    // ? . #
    if ((stones[loc].notleft()) && (stones[loc-1].notleft()) &&
        (stones[loc].nottop())) {
        return (stones[loc-2].black() &&
                stones[loc-1-Board::BSIZE].black());
    }

    // . ?
    // ? .
    // . #
    if ((stones[loc].notleft()) && (stones[loc].nottop()) &&
        (stones[loc-Board::BSIZE].nottop())) {
        return (stones[loc-1-Board::BSIZE].black() &&
                stones[loc-2*Board::BSIZE].black());
    }

    // ? .
    // . ?
    // # .
    if (stones[loc].notright() && stones[loc].nottop() &&
        stones[loc-Board::BSIZE].nottop()) {
        return (stones[loc+1-Board::BSIZE].black() &&
                stones[loc-2*Board::BSIZE].black());
    }

    // . ? .
    // # . ?
    if (stones[loc].nottop() && stones[loc].notright() &&
        stones[loc+1].notright()) {
        return (stones[loc+2].black() &&
                stones[loc+1-Board::BSIZE].black());
    }

    // # . ?
    // . ? .
    if (stones[loc].notright() && stones[loc+1].notright() &&
        stones[loc].notbottom()) {
        return (stones[loc+2].black() &&
                stones[loc+1+Board::BSIZE].black());
    }

    // # .
    // . ?
    // ? .
    if (stones[loc].notright() && stones[loc].notbottom() &&
        stones[loc+Board::BSIZE].notbottom()) {
        return (stones[loc+2*Board::BSIZE].black() &&
                stones[loc+1+Board::BSIZE].black());
    }

```

```
    // . #
    // ? .
    // . ?
    if (stones[loc].notleft() && stones[loc].notbottom() &&
        stones[loc+Board::BSIZE].notbottom()) {
        return (stones[loc-1+Board::BSIZE].black() &&
                stones[loc+2*Board::BSIZE].black());
    }

    // ? . #
    // . ? .
    if (stones[loc].notleft() && stones[loc-1].notleft() &&
        stones[loc].notbottom()) {
        return (stones[loc-2].black() &&
                stones[loc-1+Board::BSIZE].black());
    }

    // ? . ?
    // . # .
    if (stones[loc].nottop() && stones[loc].notleft() &&
        stones[loc].notright()) {
        return (stones[loc-1-Board::BSIZE].black() &&
                stones[loc+1-Board::BSIZE].black());
    }

    // . ?
    // # .
    // . ?
    if (stones[loc].notright() && stones[loc].nottop() &&
        stones[loc].notbottom()) {
        return (stones[loc+1+Board::BSIZE].black() &&
                stones[loc+1-Board::BSIZE].black());
    }

    // . # .
    // ? . ?
    if (stones[loc].notbottom() && stones[loc].notright() &&
        stones[loc].notleft()) {
        return (stones[loc-1+Board::BSIZE].black() &&
                stones[loc+1+Board::BSIZE].black());
    }

    // ? .
    // . #
    // ? .
    if (stones[loc].notleft() && stones[loc].nottop() &&
        stones[loc].notbottom()) {
        return (stones[loc-1+Board::BSIZE].black() &&
                stones[loc-1-Board::BSIZE].black());
    }
```

```
    return false;
}
```

A.3 Exodus File Documentation

A.3.1 agent.cpp File Reference

Implementation of **Agent** (p. 73) and **AgentShell** (p. 76) classes.

Defines

- #define **LOG**(x)

  Macro for outputing to the log file.

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.1.1 Detailed Description

Implementation of **Agent** (p. 73) and **AgentShell** (p. 76) classes.

Revision:
1.21

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.1.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: agent.cpp,v 1.21 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.2 agent.h File Reference

Header file for **Agent** (p. 73) related classes.

Compounds

- class **Agent**

  Defines the basic structure of an agent.

- class **AgentShell**

  Represents a single thread in a thread pool.

- class **ExtenderAgent**

  Suggests moves that extend from friendly stones.

- class **FollowerAgent**

  Suggests moves near opponent's last move.

- class **GroupStatsAgent**

  Agent (p. 73) *to calculate group information.*

- class **OpenerAgent**

  Suggests good opening moves.

- class **RandomAgent**

  Suggests random legal moves.

- class **TigersMouthAgent**

  Tries to make tiger's mouths.

Defines

- #define **MAX_AGENTS** 5

    Maximum number of agents allowed.

- #define **MAXSTONE** 10

    Highest logical value for probability board element.

- #define **MAXGROUPS** 50

    Maximum number of distinct groups.

A.3.2.1 Detailed Description

Header file for **Agent** (p. 73) related classes.

This file contains headers for the **Agent** (p. 73) class, the **AgentShell** (p. 76) class, and all of the individual agents

Revision:
1.22

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

A.3.2.2 Define Documentation

#define MAXSTONE 10 Highest logical value for probability board element.

All probability boards generated by all agents shall output a value of MAXSTONE for highly suggested values and 0 for unsuggested values. No agent shall make an element of a probability board larger than this value.

A.3.3 bdemo.cpp File Reference

Prints a demo board for numerical reference.

Functions

- int **main** (int argc, char *argv[])

  Main function for printing demo boards.

A.3.3.1 Detailed Description

Prints a demo board for numerical reference.

Revision:
1.5

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.3.2 Function Documentation

int main (int *argc*, char * *argv*[]) Main function for printing demo boards.

Demo boards are just ASCII representations of the goban that has at each location of the board the number representing the offset into a single dimention array. For example, a 9x9 board's leftmost value for the second row from the top is "9." The third row from the top would be "18."

```
                                 {
    if (argc == 2) {
      int val =  atoi(argv[1]);
      if (val < 1)  {
         cout << "Invalid board size." << endl;
      } else {
         print_demo(val);
      }
    } else {
      cout << "Please give board size as single parameter."
           << endl;
    }
    return 0;
 }
```

A.3.4 blackboard.cpp File Reference

Implementation of **Blackboard** (p. 77) class.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.4.1 Detailed Description

Implementation of **Blackboard** (p. 77) class.

Revision:
1.9

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.4.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: blackboard.cpp,v 1.9 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.5 blackboard.h File Reference

Header file for the **Blackboard** (p. 77) class.

Compounds

- class **Blackboard**

  This class contains globaly relavent information.

A.3.5.1 Detailed Description

Header file for the **Blackboard** (p. 77) class.

Revision:
1.9

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

A.3.6 board.cpp File Reference

Implementation for **Board** (p. 78) class.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Functions

- ostream& **operator**<< (ostream &strm, **Board** &aBoard)

  Output operator.

Variables

- char **rcsid** [ ] = "$Id: board.cpp,v 1.22 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.6.1 Detailed Description

Implementation for **Board** (p. 78) class.

Revision:
1.22

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.7 board.h File Reference

Header file for board class.

Compounds

- class **Board**

  Defines a goban abstraction.

Typedefs

- typedef **usi_t loc_t**

  Offset into board 1D array.

Functions

- ostream& **operator**<< (ostream &strm, **Board** &aBoard)

  Output operator.

A.3.7.1 Detailed Description

Header file for board class.

Revision:
1.14

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.8 config.h File Reference

System configuration definitions.

Defines

- #define **STDC_HEADERS** 1
- #define **HAVE_FCNTL_H** 1
- #define **HAVE_MALLOC_H** 1
- #define **MAX_THREADS** 10

  Maximum number of threads that can be active at the same time.

- #define **MAX_AGENT_THREADS** 5

  Maximum number of threads in the thread pool. This value only includes threads used by the agents and does not include other supplemental threads.

- #define **AGENTS_USED** {"**RandomAgent**", "end"}

  Agents used in this compilation of the program.

- #define **PERLTK** 1

  This tells that the Perl/TK GUI will be used.

- #define **NDEBUG** 1

  This deactivates assertions.

A.3.8.1 Detailed Description

System configuration definitions.

Warning:

Automatically generated by ../configure script

A.3.8.2 Define Documentation

#define HAVE_FCNTL_H 1 Define if you have the <fcntl.h> header file.

#define HAVE_MALLOC_H 1 Define if you have the <malloc.h> header file.

#define MAX_THREADS 10 Maximum number of threads that can be active at the same time.

This value includes all threads running.

#define STDC_HEADERS 1 Define if you have the ANSI C header files.

A.3.9 dummygenerator.cpp File Reference

Implementation of random move generator called **DummyGenerator** (p. 88).

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ]
  Source code identifier.

A.3.9.1 Detailed Description

Implementation of random move generator called **DummyGenerator** (p. 88).

Revision:
1.11

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.9.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: dummygenerator.cpp,v 1.11 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.10 exodus.h File Reference

Global constants declarations.

Compounds

- struct **global_data_t**
  global data structure.

Defines

- #define **NUM_THREADS** 5

  Number of threads in the thread pool.

- #define **VERSION** "R016_000B"

  Program version.

- #define **LOG**(x)

  Log file macro.

Typedefs

- typedef unsigned short int **usi_t**

  unsigned short int.

Enumerations

- enum **color_t** { **EMPTY**, **WHITE**, **BLACK** }

  Stone (p. 160) *color type.*

Variables

- **global_data_t global_data**

  Holds the minimal amount of global data required for this program.

- ofstream **log_cout**

  Output stream for log file.

- pthread_mutex_t **log_mutex**

  MUTEX for writing to the log file.

- int **thread_count**

  Number of threads currently open.

A.3.10.1 Detailed Description

Global constants declarations.

Conventions used in this project:

1. Function braces are all in far-left
2. Loop start brace is on same line; end brace is far-left
3. Classes start with a capital letter
4. Classes with name inside have each new word in caps
5. vars use underscores
6. types end in _t
7. globals in all caps ??? maybe end in _g
8. defines in all caps
9. abbreviations:
   - tmp......temporary
   - ptr......pointer
   - func.....function
10. Functions start with lowercase letter (unless constructor, etc.)
11. Each new word in a function is delimited with underscores

Revision:
1.27

Date:
2003/04/30 01:57:59

Todo:
Agents need to be able to communicate for complex situations.

Author:
Todd Blackman

A.3.10.2 Typedef Documentation

typedef unsigned short int usi_t unsigned short int.

Used so much, this makes code neater

A.3.11 extenderagent.cpp File Reference

Implementation of an **ExtenderAgent** (p. 89) that attempts to extend from friendly stones.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.11.1 Detailed Description

Implementation of an **ExtenderAgent** (p. 89) that attempts to extend from friendly stones.

Revision:
1.6

Date:
2003/04/30 01:54:48

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.11.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: extenderagent.cpp,v 1.6 2003/04/30 01:54:48 blackman Exp $"
```

Source code identifier.

A.3.12 followeragent.cpp File Reference

Implementation of **FollowerAgent** (p. 92) which plays moves close to opponent.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.12.1 Detailed Description

Implementation of **FollowerAgent** (p. 92) which plays moves close to opponent.

Revision:
1.8

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.12.2 Variable Documentation

char rcsid [static] **Initial value:**

```
"$Id: followeragent.cpp,v 1.8 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.13 ga.cpp File Reference

Implementation for **Ga** (p. 96) class.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Functions

- ostream& **operator**<< (ostream &strm, const **Population** &pop)

  Population (p. 153) *output operator.*

- bool **strchrom** (**chromosome_t** chrom, string chrom2)

  Simple comparison function.

Variables

- char **rcsid** [ ] = "$Id: ga.cpp,v 1.28 2003/04/30 01:54:48 blackman Exp $"

  Source code identifier.

A.3.13.1 Detailed Description

Implementation for **Ga** (p. 96) class.

Revision:

1.28

Date:

2003/04/30 01:54:48

WORKS CITED:

author=David E. Goldberg title=Genetic Algorithms in Search, Optimization, and Machine Learning publisher=The University of Alabama year=1989

title=Numerical Recipes in C: The Art of Scientific Computing publisher=The Press Syndicate of the University of Cambridge edition=Second year=1997 pages=227,616–619

Author:

Todd Blackman

A.3.13.2 Function Documentation

bool strchrom (chromosome_t *chrom*, string *chrom2*) `[static]` Simple comparison function.

This function allows the programmer to compare a string representation of a chromosome with the datastructure representation.

```
{
    unsigned int x;

//cout << endl;
//cout << chrom << endl;
//cout << chrom2.length() << endl;
    for (x=0; x<chrom2.length(); x++) {
        bool b;
        b = (chrom2[x] == '1') ? true : false;
//cout << b << flush;
        if (chrom[x] != b) break;
    }
//cout << endl;
    if (chrom2.length() == x) return true;
    return false;
}
```

A.3.14 ga.h File Reference

Header file for genetic algorithm related classes.

Compounds

- class **Ga**

  Defines a Genetic Algorithm.

- struct **Individual**

  An individual in a population of a GA.

- struct **Population**

  A single population within a GA.

Defines

- #define **MAXPOP** 10000

  Maximum size of population.

A.3.14.1 Detailed Description

Header file for genetic algorithm related classes.

This file also includes some statistics functions and definitions for **Ga** (p. 96), **Population** (p. 153), and **Individual** (p. 142)

Revision:
1.16

Date:
2003/04/30 01:57:59

Todo:
Get rid of vectors and replace with arrays

Author:
Todd Blackman

A.3.15 gafunc.h File Reference

Header file for GA testing and aux. functions.

Compounds

- class **PreCodex**

  Allows other classes to profide a fitness function.

- class **testCodex**

  A testing fitness function provider.

A.3.15.1 Detailed Description

Header file for GA testing and aux. functions.

This file provides **PreCodex** (p. 154) and **testCodex** (p. 169)

Revision:
1.12

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.16 game.cpp File Reference

Implementation of the **Game** (p. 112) class.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Functions

- ostream& **operator**<< (ostream &strm, **Game** &aGame)

  Stream operator.

Variables

- char **rcsid** []

  Source code identifier.

A.3.16.1 Detailed Description

Implementation of the **Game** (p. 112) class.

Revision:
1.28

Date:
2003/04/23 21:42:59

Bug:
super-ko does not take rotation and symmetry into account.

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.16.2 Function Documentation

ostream & operator<< (ostream & *strm*, Game & *aGame*) Stream operator.

This is used to output the latest state of the game

```
                                                    {
    Board b = aGame.get_board();
    strm << b;
    return strm;
}
```

A.3.16.3 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: game.cpp,v 1.28 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.17 game.h File Reference

Header file for game class.

Compounds

- class **Game**

  A class that defines a series of boards.

Defines

- #define **SUICIDE_CHECK** 1

  Enable checking for suicide.

- #define **SUPERKO_CHECK** 0

  Enable checking for superko.

Typedefs

- typedef pair<**usi_t**, **usi_t**> **usi_p**

  Type to make code simpler.

Functions

- ostream& **operator**<< (ostream &strm, **Game** &aGame)

  Stream operator.

A.3.17.1 Detailed Description

Header file for game class.

Revision:
1.21

Date:
2003/04/30 01:57:59

Author:

Todd Blackman

A.3.17.2 Function Documentation

ostream& operator<< (ostream & *strm*, Game & *aGame*) Stream operator.

This is used to output the latest state of the game

```
{
    Board b = aGame.get_board();
    strm << b;
    return strm;
}
```

A.3.18 gatypes.h File Reference

Header file for genetic algorithm types and defaults.

Typedefs

- typedef bool **allele**

  Allele type.

- typedef vector<**allele**> **chromosome_t**

  Chromosome type.

Functions

- ostream& **operator**<< (ostream &strm, const **chromosome_t** &chrom)

  Chromosome output operator.

A.3.18.1 Detailed Description

Header file for genetic algorithm types and defaults.

Revision:
1.10

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

A.3.19 genalgogenerator.cpp File Reference

A genetic algorithm player using agents.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.19.1 Detailed Description

A genetic algorithm player using agents.

Revision:
1.31

Date:
2003/04/30 01:54:48

Author:
Todd Blackman

A.3.19.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: genalgogenerator.cpp,v 1.31 2003/04/30 01:54:48 blackman Exp $"
```

Source code identifier.

A.3.20 ginterface.cpp File Reference

Implementation of a GUI interface.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ] = "$Id: ginterface.cpp,v 1.24 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.20.1 Detailed Description

Implementation of a GUI interface.

This file provides **GUIInterface** (p. 138)

Revision:
1.24

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.21 groupstatsagent.cpp File Reference

Provides an agent to calculate group information.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.21.1 Detailed Description

Provides an agent to calculate group information.

This agent (**GroupStatsAgent** (p. 136)) calculates a unique number for each group on the board. Planned for this agent are the following tasks:

*) Assign a unique number to each group

*) Count the liberties of each group

*) Calculate a safety value

This agent also acts as a "killer" agent that tries to capture enemy stones

Todo:
Make this more memoized

Revision:
1.7

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.21.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: groupstatsagent.cpp,v 1.7 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.22 iinterface.cpp File Reference

Implementation of IGS interface class (**IGS_Interface** (p. 141)).

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ] = "$Id: iinterface.cpp,v 1.11 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.22.1 Detailed Description

Implementation of IGS interface class (**IGS_Interface** (p. 141)).

Revision:
1.11

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.23 interface.cpp File Reference

Implementation for abstract **Interface** (p. 144) classes.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

A.3.23.1 Detailed Description

Implementation for abstract **Interface** (p. 144) classes.

This file also includes all code for all abstract classes below **Interface** (p. 144), but above an actual interface or generator.

Revision:
1.16

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.24 interface.h File Reference

Header file for interfaces.

Compounds

- class **GaTrainerInterface**

  Used to train a GA to work correctly.

- class **GoModemInterface**

  Go modem interface.

- class **GUIInterface**

  Graphical User **Interface** *(p. 144).*

- class **IGS_Interface**

  Internet Go Server (IGS) **Interface** *(p. 144).*

- class **Interface**

  The interface between a move generator (outside) and the inside of the program.

- class **NNGS_Interface**

  No Name Go Server **Interface** *(p. 144).*

- class **TextInterface**

  Text **Interface** *(p. 144).*

A.3.24.1 Detailed Description

Header file for interfaces.

This file provides interfaces between the outside world and the program. The first virtual class, **Interface** (p. 144), defines the functionality of the various interfaces that the program will have to be aware of. View this first class as a two-way pipe. All interfaces have a public method called \emph{start} that spawns a thread and then returns. Other Interfaces can be found in outputgen.cpp

Revision:
1.20

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.25 main.cpp File Reference

Main, cmd-line, init-file functions.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Functions

- int **main** (int argc, char *argv[ ])

  Main function.

- void **start_training** (void)

  Start a GA run.

- void **start_play** ()

  Starts a game between GA and a human.

- void **initialize** ()

  Runs regressions Initializes some aspects.

- void **parse_cmd_line_options** (int argc, char **argv)

  Reads command-line parameters.

- void **parse_rc_file** ()

  parses .exodusrc file.

- void **assign_global** (const char *var, const char *val)

  Assign global variable values.

- void **print_welcome** (void)

  Prints a welcome message.

- void **print_help** (void)

  Prints a help message.

Variables

- char **rcsid** [ ]

  Source code identifier.

- **global_data_t global_data**

  Holds the minimal amount of global data required for this program.

- int **thread_count** = 1

  Number of threads currently open.

- pthread_mutex_t **log_mutex**

  MUTEX for writing to the log file.

- ofstream **log_cout**

  Output stream for log file.

A.3.25.1 Detailed Description

Main, cmd-line, init-file functions.

Main program, command-line parsing, init-file parsing

Revision:
1.43

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.25.2 Function Documentation

void assign_global (const char * *var*, const char * *val*) Assign global variable values.

Assign the value val to the global variable var with error checking

Parameters:

var What global variable to assign a value to

val The value to assign to the global variable specified in the other parameter

Precondition:

var and val point to valid strings.

```
                                                         {
    //cout << "-D- Doing " << var << " with " << val << endl;

    //General parameters
    if (!strcmp(var, "welcome")) {
      global_data.welcome = atoi(val);
    } else if (!strcmp(var, "train")) {
      global_data.train = atoi(val);
    } else if (!strcmp(var, "gui")) {
      GUIInterface::GPATH = val;
    } else if (!strcmp(var, "verbosity")) {
      global_data.verbosity = atoi(val);
      if ((global_data.verbosity < 0) ||
          (global_data.verbosity > 10)) {
         global_data.verbosity=0;
         cout << "-W- Verbosity must be between 0 and 10 inclusive. "
              << "Set to 0." << endl;
      }
    } else if (!strcmp(var, "version")) {
      global_data.version = atoi(val);
    } else if (!strcmp(var, "help")) {
      global_data.help = atoi(val);
//   } else if (!strcmp(var, "thread_count")) {
//     global_data.thread_count = atoi(val);
//     if ((global_data.thread_count < 1) ||
//         (global_data.thread_count > MAX_THREADS)) {
//        global_data.thread_count=1;
//        cout << "-W- Thread_count must be between 1 and "
//             << MAX_THREADS << " inclusive. "
//             << "Set to 1." << endl;
//     }
    } else if (!strcmp(var, "reg_on")) {
      global_data.reg_on = static_cast<bool>(atoi(val));

    // Go specific parameters
    } else if (!strcmp(var, "bsize")) {
      Board::BSIZE = atoi(val);

      if ((Board::BSIZE < 3) || (Board::BSIZE > 19)) {
         Board::BSIZE=19;
         cout << "-W- bsize must be between 3 and 19"
              << " inclusive. Set to 19." << endl;
```

```
    }
  } else if (!strcmp(var, "super_ko")) {
    Game::SUPER_KO = static_cast<bool>(atoi(val));
  } else if (!strcmp(var, "komi")) {
    Game::KOMI = atof(val);
    if ((Game::KOMI < -15.0) || (Game::KOMI > 15.0)) {
       cout << "-W- KOMI value is strange: " << Game::KOMI << endl;
    }
  } else if (!strcmp(var, "initial_time")) {
    Game::INITIAL_TIME = atoi(val);
  } else if (!strcmp(var, "byomi_time")) {
    Game::BYOMI_TIME = atoi(val);
  } else if (!strcmp(var, "byomi_stones")) {
    Game::BYOMI_STONES  = atoi(val);
  } else if (!strcmp(var, "suicide")) {
    Game::SUICIDE = static_cast<bool>(atoi(val));
  } else if (!strcmp(var, "my_color")) {
  } else if (!strcmp(var, "num_handicap")) {
    Board::HANDICAP = atoi(val);
    if (Board::HANDICAP > 9) cout << "-W- Handicap is strange: "
                                  << Board::HANDICAP << endl;
  } else if (!strcmp(var, "handicap_placement")) {
     cout << "-W- handicap placement not implemented yet." << endl;
  // GA parameters
  } else if (!strcmp(var, "resume")) {
    // Resume implies train
    global_data.train=1;
    Ga::FILENAME_IN = val;
  } else if (!strcmp(var, "train_file")) {
    Ga::TRAIN_FILE = val;
  } else if (!strcmp(var, "output")) {
    Ga::FILENAME_OUT = val;
  } else if (!strcmp(var, "best")) {
    Ga::BEST_FILENAME_OUT = val;
  } else if (!strcmp(var, "popsize")) {
    Ga::POPSIZE = atoi(val);
  } else if (!strcmp(var, "maxgen")) {
    Ga::MAXGEN = atoi(val);
  } else if (!strcmp(var, "fitness_cutoff")) {
    Ga::FITNESS_CUTOFF = atoi(val);
  } else if (!strcmp(var, "pcross")) {
    Ga::PCROSS = atoi(val);
  } else if (!strcmp(var, "pmutation")) {
    Ga::PMUTATION = atoi(val);
  } else if (!strcmp(var, "fmultiple")) {
    Ga::FMULTIPLE = atoi(val);
  } else {
    cout << "-W- Invalid parameter: " << var << endl;
  };
};
```

void parse_rc_file () parses .exodusrc file.

Looks for a .exodusrc file in the current directory. It parses the file if it is found.

Precondition:

None, but .exodusrc must be in current directory for it to be found.

Postcondition:

variables/parameters specified in rc file have been communicated to this program and are stored in global variables. All paths equal either the null string or a valid path to a valid file when finished.

Returns:

none

```
                    {
void assign_global(const char *, const char *);

char line[256];
char var[100], val[100];
char *ptr;

//give global variables initial values.
global_data.welcome=1;
global_data.train=1;

GUIInterface::GPATH = "";

global_data.verbosity=0;
global_data.version=0;
global_data.help=0;
//global_data.thread_count=1;
global_data.reg_on=0;

Board::BSIZE=9;

Game::SUPER_KO = false;
Game::KOMI = 0.5;
Game::SUICIDE = false;

Game::INITIAL_TIME=300;
Game::BYOMI_TIME=300;
Game::BYOMI_STONES=10;

// GA related
Ga::MAXGEN = 10;
Ga::POPSIZE = 20;
Ga::FITNESS_CUTOFF = 1.0;
Ga::PCROSS = 0.4;
```

```
Ga::PMUTATION = 0.0333;
Ga::FMULTIPLE = 2.0;
Ga::FILENAME_IN = "";
Ga::FILENAME_OUT = "ga.save";
Ga::BEST_FILENAME_OUT = "ga.best.save";
Ga::TRAIN_FILE = "ga.train";

global_data.my_color=BLACK;
Board::HANDICAP=0;
//global_data.handicap_placement=0;

//Construct string containing path to rc file (which might not be there).
//current_dir = get_current_dir_name();
//rcfile = (char *) malloc((sizeof current_dir) + 15);
//sprintf(rcfile, "%s/.exodusrc", current_dir);
//cout << rcfile << endl;

//Attempt to open the rc file and parse it.
ifstream fin(".exodusrc");
if (fin) {

  while ( !fin.eof() ) {

    fin.getline(line, 255);

    filter_whitespace(line);

    if ((line[0] != '#') && (strlen(line) != 0)) {
      //cout << line << endl;

      //Construct val value from line
      ptr = strpbrk(line, "=");
      if ((ptr != NULL) && (strlen(ptr) >= 2)) {
         ptr++;
         if (*ptr == ' ') ptr++;
         strcpy(val, ptr);
      } else {
        //default to true
        strcpy(val, "1");
      }

      //Construct var value from line
      ptr = strpbrk(line, "=");
      if (ptr != NULL) {
         *ptr = '\0';
         if ( *(ptr - 1) == ' ') *(ptr - 1) = '\0';
      }
      strcpy(var, line);

      //cout << var << " = " << val << endl;
      assign_global(var, val);
```

```
       }
     }

     fin.close();
   }
}
```

void print_welcome (void) Prints a welcome message.

Author:

Anonymous Web program (large text)

```
                         {
   cout << "===================================================" << endl;
   cout << "               ______    " << endl;
   cout << "              / ____/___ " << endl;
   cout << "             / / __/ __ \\" << endl;
   cout << "            / /_/ / /_/ /" << endl;
   cout << "The game of \\____/\\____/ " << endl;
   cout << "" << endl;
   cout << "                  ______               _           " << endl;
   cout << "                 |  ____|             | |          " << endl;
   cout << "                 | |__  __  _____   __| |_   _ ___ " << endl;
   cout << "                 |  __| \\ \\/ / _ \\ / _‘ | | | / __|" << endl;
   cout << "                 | |____ >  < (_) | (_| | |_| \\__ \\" << endl;
   cout << "A program called |______/_/\\_\\___/ \\__,_|\\__,_|___/" << endl;
   cout << endl;
   cout << "Written by Todd Blackman" << endl;
   cout << "===================================================" << endl;
   cout << endl;
   return;
}
```

A.3.25.3 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: main.cpp,v 1.43 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.26 moderator.t File Reference

Implementation and definition of **Moderator** (p. 146) template.

Compounds

- class **Moderator**

  Encapsulates two interfaces and has them play together.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsidm** [ ] = "$Id: moderator.t,v 1.20 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.26.1 Detailed Description

Implementation and definition of **Moderator** (p. 146) template.

Revision:
1.20

Date:
2003/04/23 21:42:59

Copyright 2001, 2002, 2003

Author:
Todd Blackman

A.3.27 move.cpp File Reference

Implementation of the **move_t** (p. 148) stuct.

Defines

- #define **LOG**(x)
  Macro for outputing to log file.

Variables

- char **rcsid** [ ]
  Source code identifier.

A.3.27.1 Detailed Description

Implementation of the **move_t** (p. 148) stuct.

Revision:
1.9

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.27.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: move.cpp,v 1.9 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.28 move.h File Reference

Describes a Move struct.

Compounds

- struct **move_t**

  A single move on the goban.

A.3.28.1 Detailed Description

Describes a Move struct.

Revision:
1.4

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

A.3.29 openeragent.cpp File Reference

Opening move agent.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** []

  Source code identifier.

A.3.29.1 Detailed Description

Opening move agent.

Revision:
1.16

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.29.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: openeragent.cpp,v 1.16 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.30 outputgen.h File Reference

Header file for **GenAlgoGenerator** (p. 126), **NeuralNetGenerator** (p. 150), and **DummyGenerator** (p. 88) classes.

Compounds

- class **DummyGenerator**

  A dummy move generator that generates random legal moves.

- class **GenAlgoGenerator**

  A genetic algorithm move generator.

- class **NeuralNetGenerator**

  A Neural Network move generator.

Defines

- #define **BITSPERWEIGHT** 4

  Each network weight is an integer represented in this number of bits.

- #define **SECONDLEVELNODES** 3

  Number of nodes at the second level of the network.

A.3.30.1 Detailed Description

Header file for **GenAlgoGenerator** (p. 126), **NeuralNetGenerator** (p. 150), and **DummyGenerator** (p. 88) classes.

Output generators take the agents and a blackboard and use them to generate the next move. Essentially, this class takes a move as input and outputs a move.

Revision:
1.19

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

A.3.31 probboard.cpp File Reference

The implementation for the probability board.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Functions

- ostream& **operator**<< (ostream &strm, **ProbBoard** &aBoard)

  Stream operator.

Variables

- char **rcsid** [ ] = "$Id: probboard.cpp,v 1.22 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.31.1 Detailed Description

The implementation for the probability board.

This file provides the **ProbBoard** (p. 155) class.

Revision:
1.22

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.32 probboard.h File Reference

Probability matrix for an agent's next move.

Compounds

- class **ProbBoard**

  Agent (p. 73)*'s probability output board.*

Defines

- #define **SPIN** 0

  Spin or choose first highest location on probboard.

A.3.32.1 Detailed Description

Probability matrix for an agent's next move.

Revision:
1.15

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

A.3.32.2 Define Documentation

#define SPIN 0 Spin or choose first highest location on probboard.

- If 1, then spin a wheel for choosing location on the board.
  - If 0, just choose the last location with the maximum value.

A.3.33 randomagent.cpp File Reference

Random agent implementation.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ] = "$Id: randomagent.cpp,v 1.5 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.33.1 Detailed Description

Random agent implementation.

This file contains the implementation of the **RandomAgent** (p. 159) class

Revision:
1.5

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.34 stone.cpp File Reference

Implementation of the **Stone** (p. 160) class.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Functions

- ostream& **operator**<< (ostream &strm, **Stone** &aStone)

  Overload the printing operator.

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.34.1 Detailed Description

Implementation of the **Stone** (p. 160) class.

Revision:
1.11

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.34.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: stone.cpp,v 1.11 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.35 stone.h File Reference

Header file for **Stone** (p. 160) class.

Compounds

- class **Stone**

  Defines a point (stone) on the board.

Defines

- #define **INV**(x) (x==BLACK) ? WHITE : BLACK

  Macro for inverting the color.

Typedefs

- typedef unsigned short int **stone_t**

  Stone (p. 160) *bit-map type.*

Functions

- ostream& **operator**<< (ostream &strm, **Stone** &aStone)

  Overload the printing operator.

A.3.35.1 Detailed Description

Header file for **Stone** (p. 160) class.

Anticipating that this code will be called quite a bit, the implementation in this file sacrifices clarity for efficiency. Still, it should be fairly strait forward. It's just a collection of bit-masks.

Precondition:
unsigned short int is 2 bytes or more.

Warning:
Endianess of architecture may matter here.

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.36 subthread.cpp File Reference

Implementation for abstract class **Subthread** (p. 165).

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Functions

- void* **CALL_processing** (void *tmp_obj)

  Calls processing thread.

Variables

- char **rcsid** [ ] = "$Id: subthread.cpp,v 1.14 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.36.1 Detailed Description

Implementation for abstract class **Subthread** (p. 165).

Revision:
1.14

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.37 subthread.h File Reference

Defines virtual class for a running sub-thread.

Compounds

- struct **msg_t**

  A message to or from a thread.

- class **Subthread**

  Defines a sub-thread.

- class **Subthread_test**

  For debugging.

Enumerations

- enum **msg_id_t** { **NOMSG**, **SET_GAME_PTR**, **RESIGN**, **FORCE**, **QUIT**, **ERROR**, **TURN**, **FINISHED**, **LOAD**, **UPDATE**, **NOTIFY**, **SET_BB_PTR**, **SET_PROB_PTR** }

  Message types for thread communication.

A.3.37.1 Detailed Description

Defines virtual class for a running sub-thread.

Revision:
1.13

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

A.3.37.2 Enumeration Type Documentation

enum msg_id_t Message types for thread communication.

Enumeration values:
NOMSG No message.

SET_GAME_PTR Data is a points to a game class.

RESIGN An interface wishes to resign.

FORCE Force interface to make a move "soon".

QUIT quit program (usually only recved by thread).

ERROR Some error occured.

TURN Signals that it is the receiver's turn. When sent by an interface, it signals that the turn is finished.

FINISHED **Agent** (p. 73) finished its work.

LOAD **AgentShell** (p. 76) is instructed to do work as the specified agent.

UPDATE **Board** (p. 78) has changed. Take note of this.

NOTIFY Tell agent that information is availible at bb that it might need. (experimental feature).

SET_BB_PTR Set blackboard repository area pointer.

SET_PROB_PTR Set the result location for probability board.

```
             {
   NOMSG,
   SET_GAME_PTR,
   RESIGN,
   FORCE,
   QUIT,
   ERROR,
   TURN,



   // Agent related
   FINISHED,
   LOAD,

   UPDATE,
   NOTIFY,


   SET_BB_PTR,
   SET_PROB_PTR
} msg_id_t;
```

A.3.38 testcodex.cpp File Reference

Stub code for fitness function for GAs.

Functions

- ostream& **operator**<< (ostream &strm, const **chromosome_t** &chrom)

  Output stream operator.

Variables

- char **rcsid** [ ] = "$Id: testcodex.cpp,v 1.8 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.38.1 Detailed Description

Stub code for fitness function for GAs.

This file provides **testCodex** (p. 169) and **PreCodex** (p. 154) classes

Revision:
1.8

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

Copyright 2001, 2002, 2003

A.3.39 tigersmouthagent.cpp File Reference

Implementation of tiger's mouth class.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.39.1 Detailed Description

Implementation of tiger's mouth class.

This file provides **TigersMouthAgent** (p. 173) which attempts to make a tiger's mouth which is a good formation in go.

Revision:
1.3

Date:
2003/04/23 21:42:59

Author:
Todd Blackman

A.3.39.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: tigersmouthagent.cpp,v 1.3 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.3.40 tinterface.cpp File Reference

Implementation of text interface.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Variables

- char **rcsid** [] = "$Id: tinterface.cpp,v 1.25 2003/04/23 21:42:59 blackman Exp $"

  Source code identifier.

A.3.40.1 Detailed Description

Implementation of text interface.

This file provides the class **TextInterface** (p. 170).

Revision:
1.25

Date:
2003/04/23 21:42:59

Precondition:
The class has never been instantiated

Warning:
Only two interface classes may be instantiated at a time.

Author:
Todd Blackman

A.3.41 tools.cpp File Reference

Utilities.

Functions

- float **gammln** (float xx)

  Computes gamma function.

- float **betacf** (float a, float b, float x)

Evaluates continued fraction for incomplete beta function by modified Lentz's method.

- float **betai** (float a, float b, float x)

  Computes the incomplete beta function I_x(a,b).

- void **filter_whitespace** (char line[ ])

  Removes whitespace.

- bool **odd** (int n)

  Finds if the number is odd.

- void **print_demo** (int size)

  Prints a helpful diagram.

- int **pipe_getline** (int fd, string &buf, char endchar='\n')

  Read from file descriptor to endchar.

- void **loopy** (int whichone=0, bool done=false, char *msg="...DONE")

  Prints a "waiting" rotating character.

Variables

- char **rcsid** [ ] = "$Id: tools.cpp,v 1.17 2003/04/30 01:54:48 blackman Exp $"

  Source code identifier.

A.3.41.1 Detailed Description

Utilities.

Contains function to perform simple utility operations.

WORKS CITED:

title=Numerical Recipes in C: The Art of Scientific Computing publisher=The Press Syndicate of the University of Cambridge edition=Second year=1997 pages=227,616–619

$Revsion$

Date:
2003/04/30 01:54:48

Author:
Todd Blackman

A.3.41.2 Function Documentation

float betacf (float *a*, float *b*, float *x*) Evaluates continued fraction for incomplete beta function by modified Lentz's method.

Author:
Numerical Recipes in C, page 227-8

```
{
        int m,m2;
        float aa,c,d,del,h,qab,qam,qap;

        qab=a+b;
        qap=a+1.0;
        qam=a-1.0;
        c=1.0;
        d=1.0-qab*x/qap;
        if (fabs(d) < FPMIN) d=FPMIN;
        d=1.0/d;
        h=d;
        for (m=1;m<=MAXIT;m++) {
                m2=2*m;
                aa=m*(b-m)*x/((qam+m2)*(a+m2));
                d=1.0+aa*d;
                if (fabs(d) < FPMIN) d=FPMIN;
                c=1.0+aa/c;
                if(fabs(c) < FPMIN) c=FPMIN;
                d=1.0/d;
                h *= d*c;
                aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
                d=1.0+aa*d;
                if (fabs(d) < FPMIN) d=FPMIN;
                c=1.0+aa/c;
                if (fabs(c) < FPMIN) c=FPMIN;
                d=1.0/d;
                del=d*c;
                h *= del;
                if (fabs(del-1.0) < EPS) break;
        }
        //if ((m > MAXIT) && (!global_data.reg_on)) {
        if (m > MAXIT) {
           cerr << "-E- a or b too big, or MAXIT too "
```

```
                << "small in betacf" << endl;
        }
        return h;
}
```

float betai (float *a*, float *b*, float *x*) Computes the incomplete beta function I_x(a,b).

Author:

Numerical Recipes in C, page 227

```
{
     float bt;

     if ((x < 0.0) || (x > 1.0)) cerr << "-E- Bad x value in betai()" << endl;

     if ((x == 0.0) || (x == 1.0)) {
       bt=0.0;
     } else {
       bt = exp(gammln(a + b) -
                gammln(a) -
                gammln(b) +
                a*log(x) +
                b*log(1.0 - x));
     }

     if (x < (a+1.0)/(a+b+2.0))
             return bt*betacf(a, b, x)/a;
     else
             return 1.0 - bt*betacf(b, a, 1.0-x)/b;

}
```

void filter_whitespace (char *line*[ ]) Removes whitespace.

Removes leading whitespace, trailing whitespace, and compresses internal whitespace.

Invariant:

*line will not get larger

Precondition:

line points to a null terminated string of chars

Postcondition:

line points to a null terminated string of chars that is either smaller or the same size as the original value.

Parameters:

line A pointer to a string to remove excess whitespace from

Returns:

none

Author:

Todd Blackman

```
                                    {
char *tmp;
char tmp2[256];
int offset=0;
int offset2;

//Leading whitespace skip
while ( ((line[offset] == '\t') || (line[offset] == ' '))  &&
        (line[offset] != '\0')) offset++;

tmp = &(line[offset]);

//cycle through chars, looking for two whitespaces in a row. As we go,
//copy char by char into tmp2.  When two whitespace in a row found, copy
//a single space into tmp2, and ignore rest of white space.
offset=offset2=0;
while (tmp[offset] != '\0') {

   //More than one whitespace char in a row
   while ( (tmp[offset+1] != '\0') &&
           ((tmp[offset] == '\t') || (tmp[offset] == ' '))  &&
           ((tmp[offset+1] == '\t') || (tmp[offset+1] == ' '))
         ) {
      if (tmp[offset] == '\t') tmp[offset]=' ';
      offset++;
   }

   tmp2[offset2++] = tmp[offset++];

}
tmp2[offset2] = tmp[offset];

//If last char is a space, delete it from tmp2.
if ((offset2 > 0) && (tmp2[offset2-1] == ' ')) tmp2[offset2-1] = '\0';

//Copy tmp2 into line
```

```
  strcpy(line, tmp2);

  return;
}
```

float gammln (float *xx*) Computes gamma function.

Author:

Numerical Recipes in C, page 214

```
{
  double x,y,tmp,ser;
  static double cof[6]={76.18009172947146,-86.50532032941677,
          24.01409824083091,-1.231739572450155,0.1208650973866179e-2,
          -0.5395239384953e-5};
  int j;

  y=x=xx;
  tmp=x+5.5;
  tmp -= (x+0.5)*log(tmp);
  ser=1.000000000190015;
  for (j=0;j<=5;j++) ser += cof[j]/++y;
  return -tmp+log(2.5066282746310005*ser/x);
}
```

int pipe_getline (int *fd*, string & *buf*, char *endchar* = '\n') Read from file descriptor to endchar.

Parameters:

fd File descriptor

buf Buffer to read into

endchar Flag character to stop at

Warning:

This function is blocking. It returns when either a pipe read error occurs or the end of a line is reached.

Returns:

The number of characters read from the pipe

```
{
   int x=0;
   int res;
   char tempbuf[500];

   if (-1 == (res=read(fd, static_cast<void *>(&tempbuf[x]), 1))) {
      return -1;
   }
   while (tempbuf[x] != endchar) {
     if (res == 1) {
        //cout << endl << "Read a char: " << tempbuf[x] << endl;
        ++x;
     }
     if (-1 == (res=read(fd, static_cast<void *>(&tempbuf[x]), 1))) {
        return -1;
     }
   }

   tempbuf[x+1]=0;
   buf = tempbuf;
   return x;
}
```

void print_demo (int *size*) Prints a helpful diagram.

This function prints an ASCII representation of the board of the given size.

Parameters:

size The size of the board to print

```
                          {
   cout << "    ";
   for (char x='A'; x<'A'+size; ++x) {
      if (x < 'I') {
         cout << "___" << x << "";
      } else {
         cout << "___" << static_cast<char>(x+1) << "";
      }
   }
   cout << "_";
   cout << endl;

   //Rows
   int count=0;
   for (int x=size; x>0; --x) {
      cout << setw(3) << x << "|";

      for (int z=0; z<size; ++z) {
        cout << " " << setw(3) << count++ << "";
```

```
        }
        cout << " |" << setw(2) << size - x;
        cout << endl;
    }

    //Final Row of numbers
    cout << "    ";
    for (int x=0; x<size; ++x) {
        cout << "" << setw(4) << x << "";
    }
    cout << endl;

}
```

A.3.42 tools.h File Reference

Defines useful utilities.

Defines

- #define **SIGCUTOFF** 0.01

  Level of significance for mean and variance comparisons required for acceptance.

- #define **SQR**(x) ((x) ∗ (x))

  Find the square of a number macro.

- #define **MAXIT** 100

  Statistic constant.

- #define **EPS** 3.0e-7

  Statistic constant.

- #define **FPMIN** 1.0e-30

  Statistic constant.

Functions

- float **gammln** (float xx)

  Computes gamma function.

- float **betacf** (float a, float b, float x)

  Evaluates continued fraction for incomplete beta function by modified Lentz's method.

- float **betai** (float a, float b, float x)

  Computes the incomplete beta function I_x(a,b).

- void **filter_whitespace** (char line[ ])

  Removes whitespace.

- void **print_demo** (int size)

  Prints a helpful diagram.

- int **pipe_getline** (int fd, string &buf, char endchar='\n')

  Read from file descriptor to endchar.

- void **loopy** (int which, bool done=false, char *msg="...DONE")

  Prints a "waiting" rotating character.

- bool **odd** (int n)

  Finds if the number is odd.

A.3.42.1 Detailed Description

Defines useful utilities.

Revision:
1.10

Date:
2003/04/30 01:57:59

Author:
Todd Blackman

A.3.42.2 Define Documentation

#define SIGCUTOFF 0.01 Level of significance for mean and variance comparisons required for acceptance.

A.3.42.3 Function Documentation

float betacf (float *a*, float *b*, float *x*) Evaluates continued fraction for incomplete beta function by modified Lentz's method.

Author:
Numerical Recipes in C, page 227-8

```
{
        int m,m2;
        float aa,c,d,del,h,qab,qam,qap;

        qab=a+b;
        qap=a+1.0;
        qam=a-1.0;
        c=1.0;
        d=1.0-qab*x/qap;
        if (fabs(d) < FPMIN) d=FPMIN;
        d=1.0/d;
        h=d;
        for (m=1;m<=MAXIT;m++) {
                m2=2*m;
                aa=m*(b-m)*x/((qam+m2)*(a+m2));
                d=1.0+aa*d;
                if (fabs(d) < FPMIN) d=FPMIN;
                c=1.0+aa/c;
                if(fabs(c) < FPMIN) c=FPMIN;
                d=1.0/d;
                h *= d*c;
                aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
                d=1.0+aa*d;
                if (fabs(d) < FPMIN) d=FPMIN;
                c=1.0+aa/c;
                if (fabs(c) < FPMIN) c=FPMIN;
                d=1.0/d;
                del=d*c;
                h *= del;
                if (fabs(del-1.0) < EPS) break;
        }
        //if ((m > MAXIT) && (!global_data.reg_on)) {
        if (m > MAXIT) {
           cerr << "-E- a or b too big, or MAXIT too "
                << "small in betacf" << endl;
```

```
        }
        return h;
}
```

float betai (float *a*, float *b*, float *x*) Computes the incomplete beta function I_x(a,b).

Author:

Numerical Recipes in C, page 227

```
{
     float bt;

     if ((x < 0.0) || (x > 1.0)) cerr << "-E- Bad x value in betai()" << endl;

     if ((x == 0.0) || (x == 1.0)) {
       bt=0.0;
     } else {
       bt = exp(gammln(a + b) -
                gammln(a) -
                gammln(b) +
                a*log(x) +
                b*log(1.0 - x));
     }

     if (x < (a+1.0)/(a+b+2.0))
             return bt*betacf(a, b, x)/a;
     else
             return 1.0 - bt*betacf(b, a, 1.0-x)/b;

}
```

void filter_whitespace (char *line*[]) Removes whitespace.

Removes leading whitespace, trailing whitespace, and compresses internal whitespace.

Invariant:

*line will not get larger

Precondition:

line points to a null terminated string of chars

Postcondition:

line points to a null terminated string of chars that is either smaller or the same size as the original value.

Parameters:

line A pointer to a string to remove excess whitespace from

Returns:

none

Author:

Todd Blackman

```
                                     {
        char *tmp;
        char tmp2[256];
        int offset=0;
        int offset2;

        //Leading whitespace skip
        while ( ((line[offset] == '\t') || (line[offset] == ' '))  &&
                (line[offset] != '\0')) offset++;

        tmp = &(line[offset]);

        //cycle through chars, looking for two whitespaces in a row. As we go,
        //copy char by char into tmp2.  When two whitespace in a row found, copy
        //a single space into tmp2, and ignore rest of white space.
        offset=offset2=0;
        while (tmp[offset] != '\0') {

           //More than one whitespace char in a row
           while ( (tmp[offset+1] != '\0') &&
                   ((tmp[offset] == '\t') || (tmp[offset] == ' '))  &&
                   ((tmp[offset+1] == '\t') || (tmp[offset+1] == ' '))
                 ) {
              if (tmp[offset] == '\t') tmp[offset]=' ';
              offset++;
           }

           tmp2[offset2++] = tmp[offset++];

        }
        tmp2[offset2] = tmp[offset];

        //If last char is a space, delete it from tmp2.
        if ((offset2 > 0) && (tmp2[offset2-1] == ' ')) tmp2[offset2-1] = '\0';

        //Copy tmp2 into line
```

```
  strcpy(line, tmp2);

  return;
}
```

float gammln (float *xx*) Computes gamma function.

Author:

Numerical Recipes in C, page 214

```
{
  double x,y,tmp,ser;
  static double cof[6]={76.18009172947146,-86.50532032941677,
          24.01409824083091,-1.231739572450155,0.1208650973866179e-2,
          -0.5395239384953e-5};
  int j;

  y=x=xx;
  tmp=x+5.5;
  tmp -= (x+0.5)*log(tmp);
  ser=1.000000000190015;
  for (j=0;j<=5;j++) ser += cof[j]/++y;
  return -tmp+log(2.5066282746310005*ser/x);
}
```

int pipe_getline (int *fd*, string & *buf*, char *endchar* = '\n') Read from file descriptor to endchar.

Parameters:

fd File descriptor

buf Buffer to read into

endchar Flag character to stop at

Warning:

This function is blocking. It returns when either a pipe read error occurs or the end of a line is reached.

Returns:

The number of characters read from the pipe

```
{
   int x=0;
   int res;
   char tempbuf[500];

   if (-1 == (res=read(fd, static_cast<void *>(&tempbuf[x]), 1))) {
      return -1;
   }
   while (tempbuf[x] != endchar) {
     if (res == 1) {
        //cout << endl << "Read a char: " << tempbuf[x] << endl;
        ++x;
     }
     if (-1 == (res=read(fd, static_cast<void *>(&tempbuf[x]), 1))) {
        return -1;
     }
   }

   tempbuf[x+1]=0;
   buf = tempbuf;
   return x;
}
```

void print_demo (int *size*) Prints a helpful diagram.

This function prints an ASCII representation of the board of the given size.

Parameters:

size The size of the board to print

```
                          {
   cout << "    ";
   for (char x='A'; x<'A'+size; ++x) {
      if (x < 'I') {
         cout << "___" << x << "";
      } else {
         cout << "___" << static_cast<char>(x+1) << "";
      }
   }
   cout << "_";
   cout << endl;

   //Rows
   int count=0;
   for (int x=size; x>0; --x) {
      cout << setw(3) << x << "|";

      for (int z=0; z<size; ++z) {
        cout << " " << setw(3) << count++ << "";
```

```
      }
      cout << " |" << setw(2) << size - x;
      cout << endl;
   }

   //Final Row of numbers
   cout << "    ";
   for (int x=0; x<size; ++x) {
      cout << "" << setw(4) << x << "";
   }
   cout << endl;

}
```

A.3.43 traingainterface.cpp File Reference

Implementation of Trainer class for GAs.

Defines

- #define **LOG**(x)

  Macro for outputing to log file.

Functions

- istream& **operator**>> (istream &strm, **move_t** &tmove)

  Used to read a **move_t** (p. 148) *structure from a stream.*

Variables

- char **rcsid** [ ]

  Source code identifier.

A.3.43.1 Detailed Description

Implementation of Trainer class for GAs.

This file provides the **GaTrainerInterface** (p. 123) class

Revision:
: 1.22

Date:
: 2003/04/23 21:42:59

Author:
: Todd Blackman

Warning:
: Train only on boards that start empty with black moving first. Ensure that this interface is SECOND! Not first.

A.3.43.2 Variable Documentation

char rcsid `[static]` **Initial value:**

```
"$Id: traingainterface.cpp,v 1.22 2003/04/23 21:42:59 blackman Exp $"
```

Source code identifier.

A.4 Exodus Page Documentation

A.4.1 Todo List

Class Moderator Add time-keeping code.

File exodus.h Agents need to be able to communicate for complex situations.

File ga.h Get rid of vectors and replace with arrays

File groupstatsagent.cpp Make this more memoized

Member Game::legal(loc_t) Add memoizability–> store vector of legal/not-legal that is updated as moves are made.

A.4.2 Bug List

File game.cpp super-ko does not take rotation and symmetry into account.

Appendix B: Running the Program

The program was written in ANSI C++, and simply typing ./configure in the exodus directory followed by `make` in the `exodus/src` directory will create the executable called `exodus`. Command-line options can be viewed by typing `exodus -h`. Default options can be given in a file called `.exodusrc` when this file is located in the current directory. Important options follow:

`-reg_on` Run the regressions.

`-bsize=`*x* Set the board size to *x*.

`-train=`*x* If *x* is non-zero put the program in training mode otherwise put the program in playing mode.

`-output=`*name* Sets the file that stores the latest genetic algorithm generation to *name*.

`-best=`*name* Sets the file that stores the best chromosome from the last generation of the genetic algorithm to *name*. This parameter is also used to initialize the genetic algorithm player when it plays against a human player.

`-train_file=`*name* Sets the name of the data file that contains recorded games to *name*.

`-resume=`*name* If the program has been set to training mode, then this option tells the genetic algorithm to run the GA starting with the generation specified in *name*

`-maxgen=`x Set the maximum generation to x.

`-popsize=`x Set the population size to x.

`-pmutation=`x Set the mutation probability to x.

`-pcross=`x Set the crossover probability to x.

`-fmultiple=`x Set the F multiplier (in GA) to x.

Glossary

atari The single liberty left for a group.

baduk Another name of go.

dan A high ranking.

eye An open space inside of a group of stones. Two eyes make a group unconditionally alive if they are small enough.

goban A go playing board.

good shape The abstract concept that describes a set of stones that is in a formation conducive to being able to form a living group in the future.

group A set of stones that are connected to each other by being adjacent to at least one member of the group through a line on the board (i.e., not adjacent via diagonals).

liberty The adjacent locations to a stone that contain no other stones.

ko A situation involving the possible repetition of the state of the board which is not allowed.

komi Points of compensation given to white (always the second player) to equalize the effect of having to move second when the game begins. Usually it is 0.5 or 5.5 points so that a tie is impossible.

kyu A lower ranked go player.

seki Localized stalemate situation characterized by two groups sharing at least one liberty such that if one player were to play there first, his or her group would die. Neither side should play first, and neither side gains points for the territory surrounded by the two groups *in seki*.

thickness The abstract concept that describes how much a set of stones radiates influence, usually in a specific direction.

Index